FORTH09™

FORTH09 ™

September 26, 1988

CONTENTS

FORTH09

FORTH09 is a Forth-83 Standard system for use under the OS-9 operating system. The implementation uses separate data and code areas as is standard practice under OS-9. The result of this is that although all standard words match the Forth-83 Standard definitions, there may be some operational differences between FORTH09 and other Forth-83 systems due to the split memory map. The intent has been to meet the Forth-83 requirements as closely as possible without restricting the functionality under OS-9 in any way.

This chapter will concentrate on the use of FORTH09, focusing mainly on those features that are unique to this system. Those users who are not already familiar with Forth should obtain a book on the subject from your local book store. "Starting Forth" by Leo Brodie is a good beginning text, although you need to be aware that a few of the word definitions used in that book are different from the Forth-83 definitions.

When Forth "word" names are mentioned in this document they are CAPITALIZED (unless the word is a lower case word). Word names are also **highlighted** in most cases. The complete explanation of each word's function can be found in the glossary section later in this document. Refer to the glossary for more information about any Forth word mentioned in this text.

SYSTEM INSTALLATION

The FORTH09 distribution disk contains a number of files that should be copied to your working disk. (Make a backup of the distribution disk before proceeding).

From the distribution **CMDS** directory, copy

FLOAD to your working disk execution directory.

From the distribution **F09** directory, copy

FORTH
BLOCKS
BLOCKS.S to your working data directory.

To enter the Forth system:

chd to your working data directory (containing Forth, blocks, and blocks.s)
and type:

fload Forth

This loads the core system which is saved in the data module "Forth" . To load the complete standard system type:

1 LOAD (or FORTH-83)

Be sure you are typing in upper-case characters. (All Forth words shown in upper-case in this manual must be typed in upper-case to be found in the dictionary unless you set the **VAR CAPS** to TRUE.)

The appearance of the prompt character (">") indicates that the load is complete and the interpreter is ready to accept keyboard input.

1 LOAD will load the entire "required word set" and many of the extensions, however it will not load the screen editor. If you do a 2 LOAD instead of a 1 LOAD on entry, the Color Computer 3 80 column screen editor will also be loaded. You must be operating from an 80 column window on the Color Computer 3 prior to invoking the screen editor. If you are using a video terminal instead of the Color Computer then see the EDITOR chapter on specializing the editor for your terminal. See the line editing words **L** and **R** in the glossary. These are loaded by 1 LOAD , so they are available to modify the editor if need be. NOTE: The above procedure does not load all of the words listed in the

glossary and various other parts of this manual. You may need to load additional screens for some of these words.

The second task you should undertake after installing the proper editor, (or perhaps before), is to designate the printer path for the word **SHOW**, so you will be able to print screens. The default pathname is "/P1", if this is your printer path, you need make no change. If not, then locate the screen defining word **SOPEN**, which is a local word used by SHOW to open the printer path. Edit the pathname /P1 in the definition of **SOPEN** to match your printer port. The flag **VAR 132COL?** , is used to indicate to SHOW to use the 132 column print format which lists shadow screens side by side with normal screens. If you printer path is not "/p1" then also edit this in the screen for the word **>PRINTER** .

To exit back to the OS-9 shell, type the "keyboard abort" character (break key on Color Computer). The "keyboard interrupt" character, usually ^C will break out of endless loops and return to the interpret state through the Forth word **QUIT** . All words shown as upper-case in the glossary must be typed in upper-case to be found in dictionary searches, unless the flag CAPS is set to true. CAPS is a type VAR, see glossary.

The initial " 1 LOAD " does not load all of the words listed in the glossary. You will need to search through the block storage (with **INDEX** or **LIST**) to find the appropriate blocks to load for specific words. If you try to INDEX or LIST past the end of file, the message "-- block read error" will be displayed.

PROGRAMMING MODEL

Dictionary headers in FORTH09 are kept separate from the code (i.e. all code is headerless). The code and dictionaries are further divided into **PRIMARY** and **SECONDARY** areas. The PRIMARY code area contains all code that can be saved in a stand alone executable module (application code). The SECONDARY code area contains all other code, i.e. all compiler words, defining words, editor, etc.

Each area has its own dictionary header area, thus allowing separate **FORGET**ing of PRIMARY and SECONDARY words. At any given time, definitions will only compile to one of these areas. The system VAR **PSTATE** determines whether you are "in" (i.e. compiling into) the PRIMARY or SECONDARY area. The values of **HERE** and **CURHEAD** will be different for primary and secondary areas, i.e. **HERE** and **CURHEAD** actually access different system variables depending on which area you are in. The value of CURHEAD HERE - is the available memory in that area which is used for both dictionary headers and code.

The word **ORDER** , will display the current search order, and also indicate whether definitions compile to the PRIMARY or SECONDARY areas. The word PRIMARY causes future definitions to compile to the PRIMARY area, and SECONDARY causes future definitions to compile to the SECONDARY area. A SECONDARY definition can compile any PRIMARY or SECONDARY word, but a PRIMARY definition can only compile other PRIMARY words. This is because if a primary word is saved as an executable module the secondary code will no longer be accessible since it is not a part of that module. All immediate words are secondary, but the runtime action that they compile may be primary code. Words like **IF ELSE** etc. even though they are secondary words, can be used in a primary definition since they are immediate. Attempts to compile secondary words to primary definitions will be aborted and an error message displayed.

Each vocabulary has a primary and a secondary thread, thus there is PRIMARY FORTH and SECONDARY FORTH, or PRIMARY ASSEMBLER or SECONDARY ASSEMBLER , et cetera. In fact, the **ASSEMBLER** vocabulary contains only secondary definitions, since they execute only during compile time.

The word **WORDS** will display the word names in the first vocabulary in the search order. The primary words are displayed first, followed by the secondary words.

All compiled code is 6809 machine code. Most word references are compiled as lbsr or bsr instructions, however some instructions may compile as inline machine code. (E.G. **DROP** compiles as leau 2,u ,which is faster and also shorter than an lbsr). The system VAR **COMPLIMIT** determines the maximum number of bytes that compile inline. If you set complimit equal to 6 for instance, then code words that would take 6 or fewer bytes are compiled inline (provided they are flagged with a permission bit allowing this to happen). The default setting of COMPLIMIT is 2, this provides for the shortest possible code. If maximum speed is needed for a particular word, then COMPLIMIT can be set to a higher value during the compilation of that word.

Literal values are compiled inline, and as such are faster than **CONSTANT**s, however space can be saved by using constants when they will be referenced more than once. Constants can not be legally changed in a **SAVE**d program. (This violates OS-9's rules for reentrant code). If you need to change a constant, use the type **VAR** instead.

A **VAR** is a type of **VARIABLE** that automatically fetches its value when referenced. If the variable name is preceded by the word **TO** , then a value will be stored instead of fetched.

VAR is a defining word like VARIABLE, e.g.

```
VARIABLE DOG
VAR CAT
```

to fetch the value:	DOG @
or	CAT
to store the number 3:	3 DOG !
or	3 TO CAT

Fetching a value from CAT is both shorter and faster since the word (@) need not be referenced. The store operation is about the same, but is usually one byte shorter for the VAR.

A primary VAR and primary or secondary VARIABLEs have an offset into the data space stored in their code space parameter field. At runtime the absolute data storage address is computed. A secondary type VAR actually stores its value in its code space parameter field. This is legal since the secondary code space, which only exists in the development system, not in a saved application, is actually in the FLOAD process data memory. If you save the system with the word **SAVESYS**, upon reloading that system (with FLOAD) any secondary VARs will have the value they had at the time the **SAVESYS** was done. All other variable types, including space reserved with **RMB** will be set to zero by the loader code.

BLOCK STORAGE

The block storage system utilizes "shadow" screens. A shadow screen is a comment only screen associated with a **LOAD**able screen, for the purpose of providing more documentation text. All block storage is in standard OS-9 RBF files. The LOADable screens and the shadow screens occupy separate files, with the shadow file having the same name as the LOADable but with a ".s" extension on the name. The default filename is "blocks" and "blocks.s" for the shadow screens. FLOAD can specify an alternate storage file to use instead of the default "blocks" file. The word **USING** can be invoked at any time to switch to a different file (and shadow).

Any mass storage file and its shadow may each have a maximum length of 32,768 blocks (or screens) numbered from {0..n} where n is the number of blocks-1. A file with fifty screens, will have them numbered from 0..49. To add more blank screens to a file, use the word **MOREBLOCKS**, but note that the file must be accessed first to open the file. The word **LIST** or any other word that calls **BLOCK** will open the file. To make a new empty block file, use the OS-9 BUILD command, or some other means to make a file with 0 allocated storage. Also build the shadow file at the same time. Then in FORTH09, execute **USING** to change to that file (unless you already designated that file with the FLOAD command). Perform a LIST (0 LIST) which will return a "--block read error", but this is necessary to open the file. Now use MOREBLOCKS to create some blank storage. E.G. 20 MOREBLOCKS will extend both files by 20 blank screens.

The word **SHADOW** is used to convert a LOADable screen number to a shadow screen number. The numbers are the same but with the most significant bit **OR**ed to 1 for the shadow screen. This bit flags the words **BUFFER** and **BLOCK** to access the shadow file instead of the normal file. To list the shadow screen for screen 1, 1 SHADOW LIST .

The words **EXCHANGE** , **COPY** , **SLIDE** , **CONVEY** , and **LCOPY** are used to rearrange screens. **BLK-ERASE** is used to erase a screen and its shadow. (See the glossary).

When an error occurs during the loading of a screen, the location of the error can be shown by executing the word **WHERE** .

As is the standard for Forth-83, screen number 0, is not LOADable. This is because a value of 0 in the variable **BLK** causes input to come from the console device instead.

SAVING FORTH APPLICATION AND SYSTEM MODULES

The Forth system is defined to be all Forth code present when you do an FLOAD. You may start up Forth every time with "fload forth" and then LOAD the screens you want, however there is a time saving shortcut. After you load the screens you want to use regularly, the word **SAVESYS** will save the system with all LOADed words. This can later be reloaded from the shell with FLOAD.

E.g. fload Forth
1 LOAD
SAVESYS myforth (creates file myforth in current execution directory)
OS9: fload myforth (loads system with all words that were present when SAVESYS was executed).

There is a catch however, you can not perform **SAVESYS** after a new vocabulary has been defined. Only the predefined vocabularies **FORTH** , **ASSEMBLER** , and **ONLY** may be present. The reason is that SAVESYS is really a quite complex operation, involving packing the memory map, and adjusting a number of addresses and offset parameters. Internal vocabulary pointers must be modified, and allowing for additional vocabularies would increase the already high complexity. The need for additional vocabularies is less than in some other Forth systems however because of the **LOCAL** word name feature. In fact all of the screen editor consists of local words, except for the one word **EDIT** which is placed in the FORTH vocabulary.

Application code is all code in the primary code area. You may save any PRIMARY word as a stand alone OS-9 executable module, using the words **SAVE** or **XSAVE** . All primary code up to the designated word is saved. Initialization code is generated to properly setup the registers and memory structure on entry. **BASE MEMLIMIT** and **'FREE** are initialized, and a call is made to the word you saved. When the saved application word exits, a return to the system is made through an F$EXIT system call, after resetting path 0 options. **The user must initialize all data structures.** The words **REV** and **EDITION** allow for setting the revision number and edition number of the saved modules. The word **SAVE** saves the designated word to a pathname of the same name in the execution directory. **XSAVE** allows for a pathname that differs from the word name and also allows you to designate a larger minimum data memory size for the saved module. You would use a larger minimum data memory if the runtime program will use the data between **'FREE** and **MEMLIMIT**. Note that if the application module is invoked with the shell # parameter to supply more memory, then this extra memory will be reflected in the values of **'FREE** and **MEMLIMIT**. Refer to the glossary for details on any of the words mentioned in this text.

NOTE: The only reason for the existence of separate PRIMARY and SECONDARY code areas, is to simplify the **SAVE** of the application module. If you are not planning to save a word as an executable module, then there is no reason for using the PRIMARY area. The saved PRIMARY application word, will never include the Forth dictionaries, or **BLOCK** mass storage access. Any words dealing with the dictionary in any way MUST be defined as a SECONDARY word. This includes any IMMEDIATE word, since it is only the finding of the precedence bit in the dictionary for that word that makes it IMMEDIATE. Some words defined orginally as SECONDARY can be changed to PRIMARY by editing the screen where they are defined, if you should want to use them in an application. E.g. **>PRINTER**

INPUT/OUTPUT

When FLOAD sets up the Forth system it also performs a I$SETSTT call on path 0 to change the path options. Echo is shut off and the EOF character is set to null. No other changes are made. These changes are necessary in order to make the actions of the word **KEY** match the standard. A SAVEd application module also has initialization code to change these path options. If the normal exit is made from the application module (i.e. the entry word is exited), then these options are reset on exit. It may be desirable for the application to leave these options on, if for example you wish to use the word **READLN0** for console input which is more powerful than **EXPECT**. Leaving echo on will allow for normal OS-9 line editing features. You can force the initialization code of your SAVEd module to NOT tamper with the path 0 options, by setting bit 7 of REV to a one before invoking **SAVE** (or **XSAVE**). Only the least significant 4 bits of REV {0..3} are actually used as module revision number. Bit 7 acts as a flag to bypass the path 0 options change code.

Most of the OS-9 system call words are provided. These are all PRIMARY words, and are arranged in sets, so you would only load what are needed. Most of the I/O calls are grouped together. E.g. **I$CREATE I$OPEN I$READ** etc.

The glossary defines the stack parameters for each call, but you should also refer to the OS-9 technical manual for a detailed functional description of each call. Each call has a uniform error return procedure. After each call, if an OS-9 system error was returned by the call, then the VAR **ERROR#** will contain the error number returned. If there was no error, then **ERROR#** will be zero. This allows you to either check for errors after a call or ignore them. Any register parameters used as input by the call are taken from the data stack. Any parameters returned in registers by the call, are returned on the data stack. In a number of cases it is usual to ignore some of the return parameters, or use a default for the input. Several I/O words are provided with reduced parameter usage to simplify I/O programming.

These are: **GETCH PUTCH READ WRITE READLN WRITELN READLN0**

The more commonly used **I$GETSTT** and **I$SETSTT** calls are provided as individual words. e.g. **GET.OPT SET.OPT GET.SIZE GET.POS** etc.

STRINGS

Many of the OS-9 system calls require an address of a string that is either terminated by having the high bit of the last character set, or terminated by a null, space, or other separator character. Several string words are provided to accomplish this.

See the words: **FCS" 0" " CR"**

The words **"** (quote) and **CR"** can be used in both compilation and execution mode, and produce the address and count of a text string.

E.g. " Hello there"

Leaves two numbers on the stack. The first is the address of the string, and the second is the string length in bytes (or characters). These two numbers are suitable for the word **TYPE**. The OS-9 I/O word **WRITELN** is set up to take its buffer address and byte count in this order to simplify writing a literal string to a path.

E.g. " Hello there" 1 WRITELN

Writes the string out to path number 1 (console screen, unless redirected).

The internal string storage format for **"** and **CR"** is a counted string. That is, one byte is used to store the string length count {0..255}, and the following bytes are the actual string bytes. The operation DROP 1- will convert the address count parameters on the stack to a "counted string address" as used by various system words, e.g. **WORD** returns a counted string address.

FCS" and **0"** each produce a string containing all characters after the first space after the word, up to the closing quote. Each of these leaves only an address on the stack, which is the address in memory of the first byte of the string. The usage of these is mainly with the various OS-9 system calls that require a string address, for a pathname, or module name.

FCS" produces a string with the highest bit of the last character set (similar to the FCS assembler directive). **CR"** produces a string with an ASCII carriage return character appended on the end, which is needed by the **SHELL** word and **F$FORK** . **0"** produces a string with an ASCII null character appended to the end. For most system calls using a pathname string, any of these three can be used. The **FCS"** will produce a string one byte shorter than the other two, since there is no terminating character appended, however this word is defined as compilation only. For interactive use, **0"** and **CR"** can be used since these work in both compilation and execution mode.

All of the string types covered here are literal constant strings. They may not be altered in memory once they are defined since they occupy the OS-9 module code space. You could alter the strings at runtime, but doing so violates one of the rules for reentrant modules, and makes the code non-reentrant. Some systems may also have implemented memory protection hardware that will prevent you from doing so.

To alter a string at runtime, it should be copied to a predefined data memory buffer.

E.g. 80 RMB STRINGBUF defines an 80 byte data memory buffer

" Hello OS-9" STRINGBUF SWAP CMOVE

Moves the string into the buffer just defined.

For extensive string operations, it is probably best to make up a new string defining word and structure that keeps track of the current string length. A set of words can then be produced to do whatever string manipulations you require (concatenation, comparison, etc.). The string memory can be allocated with **RMB** or by a new defining word with **ALLOT_D** .

NUMERIC INPUT

All numbers input in FORTH09 are assumed to be 16 bits in length, unless designated otherwise. If a number being input from the console, or mass storage is prefixed with the character **"#"**, then it will be converted as a double number. Any number or double number may be prefixed with a "-" (minus sign) to make it negative. A decimal point will end the number conversion (i.e. is not allowed). The usage of a decimal point is reserved to indicate floating point (not currently implemented.)

Number conversion is always in the current number base, as indicated by the value of the variable **BASE** . The words **DECIMAL** and **HEX** set the value of **BASE** to 10 and 16 respectively. Any number base between 2 and 72 may be used, but DECIMAL and HEX account for most usage.

Any number input while in execution mode will be placed on the stack. If a number is entered as part of a colon definition (i.e. compiling) then a numeric literal will be compiled such that at execution time the number will be placed on the stack.

The internal storage format for both single and double numbers is with the most significant byte at the lower memory address, and the least significant byte at the highest memory address. This is the reverse of the format internally used by Z80, 6502, and Intel processors, but is the norm for all Motorola processors.

LOCAL WORD NAMES

FORTH09 implements local word names. A local word is one which has its header removed from the dictionary when no future reference to the word is required. The words **LOCAL GLOBAL** and **MODULE** define the scope of the local name and remove the header from the dictionary. Once the header is removed, the dictionary is packed, thus saving memory and the word is no longer found in dictionary searches. The object code for the word still remains however. The word LOCAL marks the start of a group of local word definitions. GLOBAL marks the end of the local section and the start of GLOBAL definitions. The word MODULE packs the dictionary removing all the local names. No change of vocabulary, or change between PRIMARY and SECONDARY is allowed between LOCAL and MODULE. The screen editor uses these words to remove all of the editor words from the FORTH vocabulary except for the word **EDIT** . This eliminates the need for a separate EDITOR vocabulary to hide these words, and also saves memory space. The word **HIDE** provides a means to remove single word names (i.e. make them local) after they are defined. There must have been no definitions into any vocabulary, other than the one the hidden word is in, since the definition of the word to be hidden. If there have been, then invoking HIDE will mutilate the dictionary chain.

WORD WIDTH

The maximum word name length allowed is 63 characters, using a longer name than this may cause incorrect operation. (Due to the fact that the length is smudged during compilation by ORing 64 to the count). The number of characters of the name that will actually be stored in the dictionary for a new definition is determined by the value of the VAR **WIDTH**. E.G. if you set WIDTH to 3, then only the first three characters of word names will be stored for new definitions. The actual length is known also. So if you defined the words NEWXXX and NEWYYY they would both be stored as NEW??? which you would think would cause them to be confused for one another during dictionary searches. Normally this would be the case, but FORTH09 also stores a 16 bit hashing code for each name. It is extremely rare that two words would have the same hash code, even for the case given above. Since the hash code is the first thing compared during a search, the two words NEWXXX and NEWYYY would still be found as unique words. The default setting of WIDTH is 16, you may want to change this to something smaller to conserve space if you are running short. The minimum would be a WIDTH of 1, which will work fine, but makes the **WORDS** list a little hard to read.

WORD SYNONYMS

Largely due to my own personal preference, synonyms for a number of the standard words, are used throughout the blocks file with FORTH09 instead of the standard words. These are **ENDIF** instead of **THEN** (makes the IF statement a little easier to read doesn't it), and **D@ D! DDROP DDUP DOVER DROT DCONSTANT DSWAP DVARIABLE** instead of the same word names with the "2" prefix. This makes things more consistent, by having ALL words dealing with double number operands start with the letter D. To my thinking a word like **2DUP** implies doing **DUP DUP** which is not at all the same as **DUP**ing a double number is it. The "tradeoffs" used by the standards team indicate they place a higher priority on "Naming clarity" than on "historical continuity", so I rather feel they fell down on the job with these. I know I will probably hear a lot of argument about this, but I don't really want to listen to it. If you don't like it, go ahead and edit it back to the standard and don't call me. After all that's one of the nice things about Forth, isn't it. You can change things to your liking. At any rate the word **EQU** is used to make a word synonym, i.e. another dictionary entry is produced that points to the already existing code of an existing word. See EQU in the glossary.

SPECIAL CONSIDERATIONS for DEFINING WORDS

Because of the memory map used in FORTH09, you will need to give special consideration to defining words. You can use the words **CREATE** and **,** (comma) to make tables of constant values. These values should not be changed once they are stored. **ALLOT** allocates space in the code area of the memory map. Another word, **ALLOT_D** is used to allocate space in the data memory area. Any variable data space required should be allocated with **RMB** . For large variable size buffers, use **'FREE** and **MEMLIMIT** to determine the available unallocated data space address limits. These

values are calculated at runtime so they always provide the correct absolute address of available data memory.

When making a defining word with **CREATE** and **DOES>** , the defining word itself is normally a SECONDARY word. This is because the dictionaries do not exist in the application code, only when the complete system is present. However it is often desirable for the words defined by the new defining word to be PRIMARY (application) words. For this reason the runtime code after **DOES>** is always placed in the PRIMARY code area. **DOES>** makes the switch from secondary to primary automatically, and switches back at the end of the definition. This allows words defined by the new defining word to be either PRIMARY or SECONDARY words.

A new companion word to **DOES>** is **DOES>S** which has the same action but the runtime code after **DOES>S** remains in the SECONDARY code area. This is used for defining words that will only define other SECONDARY words.

The word **;CODE** functions like **DOES>** except that it defines a runtime action of a defining word in **ASSEMBLER**. The code between **;CODE** and **END-CODE** will automatically be stored in the PRIMARY code area even though the word definition is in the SECONDARY area. To make the runtime code in the secondary area, use **;CODES** instead.

An alternate method of defining runtime action for a compiling word rather than to use **DOES>** or **;CODE** , is to define the runtime action as a separate word. E.g. if the runtime action is a word called "RUNTIME" then the defining word can contain the sequence: **COMPILE RUNTIME** .

Since the compilation of words in definitions is done by compiling a bsr/lbsr instruction to the word, you can not compile a word with ' and , "tick and comma" like you can in systems that thread the addresses. The word **[']** will compile a constant that will be computed at runtime into the absolute code address, but here again you can not use , "comma" to compile that address. Instead use the word **H'** "h-tick" to find the header address of the word and then **(COMPILE)** to compile that word. **(COMPILE)** also takes care of deciding whether to compile the code inline or as a bsr or lbsr.

If you create a table of constant values using **CREATE** and , ("comma") in execution mode, the name you CREATEd will not be accessible in the dictionary until the next colon definition is complete unless you execute the word **(;)** .

E.G. CREATE SMOOZ 10000 , 1000 , 100 , 10 , 1 , (;)

The word **(;)** "unsmudges" the name SMOOZ and updates internal pointers, making SMOOZ accessible.

Variable arrays should be built with the word **RMB** .

E.g. 100 RMB BIG (creates an 100 byte array)

: BIGN 2* BIG + ;

The word **BIGN** expects an array subscript on the stack {0..49}. This is multiplied by two, to adjust for two byte elements, and added to the base address returned by **BIG**. The result is the address of the specific array element, on which you can use the standard access words @ and ! .

COMPILER ERROR MESSAGES

The compiler does error checking whenever possible. Things like imbalanced control structures and illegal nesting are trapped and reported. This should reduce the instances of system crashing and make initial debugging quicker. It should be noted however that just the existence of the ! ("store") operator makes it impossible to completely crash proof the system with out resorting to elaborate checking of each operation. This is considered undesirable due to the additional overhead required and the restrictions it would place on the user. All this boils down to saying that while "C" an some other languages let you shoot yourself in the foot, Forth will let you shoot yourself in the head. So think before you pull the trigger (i.e. hit return). It is interesting to note that most early versions of Forth contained no error checking what so ever. The current common practice is to provide some

compile time checks for imbalanced control structures and the like, but no runtime checking (which would restrict execution speed).

Error messages are preceded with "--" (two hyphens). Compile time stack checking is done for both overflow and underflow. No runtime error checking is done. However, when testing words in interpret mode, if they underflow or overflow the data stack, this will be reported. After any message is printed the word ABORT is executed, which will reset the stacks, and make the input source the console.

The exception to this is the warning: **--redefined**
which will be displayed preceded by a word name any time a word is redefined. If a new definition is made for a word that already exists in the current search order, then the --redefined warning message will be displayed. No **ABORT** is taken for this as it is a warning only.

If an error occurs while LOADing a screen, the word **WHERE** will tell which screen the error was in, and print the contents of the screen up to the point where the error was detected. If the error was a missing control word, or the like, the error will not be detected until the end of the word definition in which the error occurred.

OPTIMIZING FOR SPEED

FORTH09 was designed to make it as short an fast as possible. In most cases the shorter code is also the faster code. FORTH09 maintains several flag bits in the dictionary header for each word in addition to the customary **IMMEDIATE** precedence bit. This allows it to copy the code for some **CODE** words inline into new definitions instead of compiling it as a branch to subroutine instruction. The shortest code generated to compile any word is 2 bytes (for a short relative branch). In most cases 3 bytes will be compiled for a long relative branch to subroutine. Many of the primitive CODE words take only 2 to 8 bytes of code to implement so a significant speed improvement can be had be compiling inline code in these cases.

The VAR **COMPLIMIT** is used to determine whether or not a code word should be compiled in line. The default setting (and minimum value) of COMPLIMIT is 2. By setting COMPLIMIT to 6 or 8, you will case many of the primitive words to compile inline. E.G.:

8 TO COMPLIMIT

sets the limit to 8 bytes. In this case CODE words of 8 bytes or less will be compiled as inline code (i.e. the machine instructions are copied in line). After compiling the words you need to speed up, set COMPLIMIT back to a smaller value. Compiling words much larger than 6 or 8 bytes in line, takes up an inordinate amount of memory without increasing the speed much.

The usual practice is to debug your code first. After it runs correctly, identify specific words that need to be faster and optimize those.

Additional speed can be had by making use of Direct Page variables. The 6809 processor has the ability to access data with an 8 bit (direct page) address instead of a 16 bit address. This allows a shorter and faster instruction to access a 256 byte area with the upper 8 bits of the 16 bit address coming from an internal "Direct Page" register. Each OS-9 process has its own direct page at the start of its data area. Part of the FORTH09 direct page is used by predefined variables like **BASE** and **STATE**. The remainder (about 190+ bytes) is available for user **VARIABLE**s, **VAR**s and any space assigned by **RMB**. To take full advantage of the speed offered by the direct page you will need to use the special operators to access any direct page storage. These operators are: **V! V@ ++ += --** . See the glossary for details on these. These special operators also work with variables stored outside the direct page, but the greatest speed increase and the most compact code results from using them with variables in the direct page.

To define a direct page variable, you need only place it at the beginning of the program. The VAR **doff** (note lower case) contains the data offset used for variable storage. If the value of **doff** is **less than** 100 (hex) at the time the VAR or VARIABLE is defined, then that variable will occupy the direct page storage area. Unless you are using the special operators mentioned above on the VARs or

VARIABLEs, there will be no advantage to a variable being in the direct page. If you are using these, then give some planning to placement of variables in your program. You would not want to use a statement like (500 RMB BIGBUFFER) at the beginning because this would allocate all of the direct page as part of BIGBUFFER and not leave any for smaller variables. Generally you should define your most accessed VARs or VARIABLEs at the beginning of your program.

Because of 6809 processor architectural reasons the word **SWAP!** is faster than the word **!** (store). **SWAP!** (swapstore) performs the same operation as the two words **SWAP !** . So for fastest storage to VARIABLEs (outside of using the operator **V!**) use **SWAP!** when possible instead of !.

E.G. VARIABLE COW
COW 7 SWAP! is faster than 7 COW !

BRANCH and **?BRANCH** code which is used by the conditionals **IF** and **WHILE** and **UNTIL** generates an inline test and a short relative branch whenever possible. This takes only 4 bytes for each instance and is as short as or shorter than any other strategy. It also results in higher speed than any other strategy. If the branch is out of range (further than 127 bytes away), then code to deal with the long branch is generated instead. The long branch code used does not take up any more space, but it is slower. Usually, about the only way to end up with a long branch is if you use string literals inside the word which eat up the length of the string in bytes plus some.

If you find you need an IF statement to test for a FALSE condition only, (e.g. 0= IF), then use the word **IFF** ("if-false") instead. This will save the compilation of **0=** and the result will be faster.

SEARCH ORDER

The dictionary search order can be listed with the word **ORDER**. This also shows whether definitions will be compiled to the PRIMARY or the SECONDARY area. The PRIMARY thread of the first vocabulary in the list is searched first, then its SECONDARY thread. This process is repeated for each successive vocabulary in the list. The word **:** automatically makes the compilation vocabulary the first one in the search order, but certain immediate words used in the definition may change this. For example the word **[A** executes **ASSEMBLER** making it the first vocabulary in the search order, in which case the compilation vocabulary may not even be in the search order at all.

The vocabulary search list (pointed to by **CONTEXT**) consists of a stack of up to 7 vocabularies. The word **ONLY** clears the list leaving only the **ONLY** vocabulary at the beginning of the order. Executing a vocabulary name (e.g. **FORTH**) replaces the first name in the list with that vocabulary. Executing **ALSO** shifts the list down one postion such that the first vocabulary is duplicated. Executing another vocabulary name then places that vocabulary in the first postion, overwriting the duplicated first name. The word **SEAL** removes all occurances of the **ONLY** vocabulary.

E.g. ONLY FORTH ALSO ASSEMBLER (use **ORDER** to view the order)

DEBUGGING

One of the strengths of Forth is that it is usually an easy process the quickly test each word as you develop it. This is largely due to the use of the stack to receive arguments and pass resulting operands. It is vital that each word handles the stack as you intended. A word taking too many or not enough items from the stack, or leaving the wrong number, spells disaster. Make sure each word you create uses the stack properly. The word **.S** provides a non-destructive way to view the stack contents which is very valuable for debugging. For tough cases, the word **BREAK** may be inserted in a definition. When **BREAK** is executed during execution of the word containing it, the words execution is suspended at that point and you are returned to the interpreter. The message "-break-" displays when the **BREAK** is executed. At this point you may examine the stacks and or execute other words, and then resume execution after the **BREAK** by executing the word **GO**. If you do anything to cause and error (so **ABORT** gets executed) then you will not be able to resume execution of the word. **BREAK** is defined as a SECONDARY word, so any word you want to compile it to must

be secondary also (at least temporarily). When debugging is complete, you should of course remove **BREAK** from the words definition.

The word **.RS** can be used to view the contents of the return stack during a break. You should keep in mind however that much of what is on the return stack are the return addresses from various words being executed and not parameters you placed there. Normally the only numbers on the return stack other than return addresses are the index and limit parameters for **DO** loops, and anything you move there with **>R** .

When using **>R** and **R>** within a definition you must be absolutely sure the return stack is balanced at the end of the word definition. Whatever is on the top of the return stack at the end of the word is used as the address of the next word to execute (return address for subroutine). If the return stack is imbalanced at the end of the word (or when **EXIT** is executed), then execution will continue at what ever address was on the stack. In this case you system will crash IF YOU ARE LUCKY.

DO loops use the return stack to hold the index and limit values so the return stack must be at the same level at **LOOP** as it was at **DO** . If you use **>R** inside a do-loop then you must use **R>** before the end of the loop to balance the return stack.

The word **LOAD** also makes use of the return stack to save the source of the previous input stream, so **LOAD** can not be used inside a do-loop.

OS-9 INTERFACE

You should make a thorough study of the OS-9 technical manual to understand the use of the various system calls. The I/O calls will probably be the most used. Make sure you understand the concepts of pathnames, path numbers, standard paths, and redirection of paths.

FORTH09 with its interactive environment provides the quickest access to the complete power of OS-9 of any language. If you look at the definition for **>PRINTER** and **>SCREEN** you will see how really trivial of a chore something as useful as I/O redirection is under OS-9. These words were defined as SECONDARY, but you may easily edit their screen and compile them as PRIMARY if you want to use them in a **SAVE**d program.

You may run any OS-9 program from within the FORTH09 environment by using the word **$** .

e.g. $ DIR (runs dir command)

To do the same from within a word definition, use the word **SHELL** .

e.g. : DIR CR" DIR" SHELL ;

Creates the Forth word **DIR** which executes the OS-9 dir command by calling the shell command.

Examples of OS-9 I/O:

VAR PATH# define a VARiable to hold path number

0" myfile" 3 I$OPEN TO PATH# DROP

attempts to open the existing file "myfile" in "update" mode (3), saves the path number in PATH# for later use and DROPs the string address pointer returned that we have no further use for.

80 RMB BUF defines an 80 byte data buffer for I/O

80 BUF PATH# I$READLN DROP

Reads a line from the open path to **BUF**, then drop the actual count read. The maximum read was 80 bytes, the actual number would be less if an ASCII carriage return was encountered before 80 bytes were transferred. If you wanted the actual number read, you could omit the DROP and do further processing.

ERROR# IF etc. could be added to test for errors after any **I$** or **F$** word.

#0 PATH# I$SEEK rewinds the path to the beginning.

CR" Hello out there in the file" PATH# WRITELN

Writes the string with a carriage return on the end to the path. **WRITELN** functions similar to **I$WRITLN** except for the order of input parameters and not returning any actual count of bytes written.

PATH# I$CLOSE Closes the path. Any open paths will automatically be closed by the F$EXIT system call which is executed when the FORTH09 Interpreter, or SAVEd program exit back to the system.

The attribute and permission numbers used by the OPEN and CREATE calls, correspond to the file attribute bits on each file. E.g. 1=read, 2=write, 3=update, etc. as shown in the OS-9 technical manual.

For the **I$CREATE** call, the permission parameter will be the permission attributes attached to the new file which determine future access modes allowed. I.e. user r/w public r/w etc. The access mode parameter is the current mode in effect while the file is open. E.g. You may give the file read and write permission (3), but access it after **I$CREATE** as write only (2).

Extended Memory Usage under OS-9 Level II

There are four LEVEL II only system calls that allow usage of memory beyond the process's 64K allocation. These are **F$ALLRAM F$DELRAM F$MAPBLK** and **F$CLRBLK** . The OS-9 technical manuals have indicated that these are system mode only calls, i.e. may only be used by a system module (driver, etc.). This is incorrect. These calls are allowed in user mode in most if not all systems.

The "block" referred to by these calls, is the memory block mapped by the system hardware MMU (Memory Management Unit). The size of this block is 8k on the Color Computer 3, and 4K or 2K on most other systems. The size is system dependent on the MMU design.

F$ALLRAM is used to reserve a number of consecutive blocks of free ram for future use. Unless you actually need more than one block to be contiguous, it is best to use this call repeatedly, allocating one block at a time. Keep a table of the block numbers allocated.

F$MAPBLK is used to map a specific block into the current process's memory map. The current process must have space in its memory map first. If the process already has a full 64K map, then there will be no room to map in the additional blocks. **F$MAPBLK** will return the address within the requesting process's memory map, where the requested block was mapped. The memory may now be accessed at that address.

F$CLRBLK is used to unmap a block from the current process's memory map. This is the reverse of **F$MAPBLK**. A number of blocks can be reserved with **F$ALLRAM**, then mapped into the processes memory with **F$MAPBLK** one at a time. To access a different block, unmap the first with **F$CLRBLK** and map in the next with **F$MAPBLK**.

When finished with all external ram, use the **F$DELRAM** call to deallocate each block that was reserved with the **F$ALLRAM** call. This returns that memory to OS-9's free memory pool.

OS-9 requires a certain amount of memory for file buffers and other uses. If you allocate every last block, your process may not be able to run due to the system running out of memory to use for file buffers etc.

FORTH-83 REQUIRED DOCUMENTATION

SYSTEM DICTIONARY SPACE USED

Varies depending on what is loaded. A good approximation can be made by doing a SAVESYS of the system in question and looking at the file size of the saved file. This will be only slightly larger than the actual space used.

APPLICATION DICTIONARY SPACE AVAILABLE

This can be determined by subtracting the value of **HERE** from the value of **CURHEAD** in each of the PRIMARY and SECONDARY areas. The sum of the two numbers is the total dictionary (and code) space available, although an actual **SAVE**able "application" has only the PRIMARY space available.

DATA STACK SPACE

The default data stack size is 256 bytes. This can be altered when loading the system with an option to FLOAD .

RETURN STACK SPACE

The default return stack size is 256 bytes, and should be the minimum used. This can be altered when loading the system with an option to FLOAD .

MASS STORAGE BLOCK RANGES

The BLOCKS file that contains the system source uses blocks 0 through 60+ (use INDEX to determine the upper limit used). Any blank block numbers are available to the user, and the file may be extended up to 32,768 blocks or the constraint of the media. It is recommended however that user code be placed in separate files. See preceding section on block storage.

OPERATOR'S TERMINAL FACILITIES

In interpretation mode the normal OS-9 line editor functions are active on the console input. Word definitions typed in may span more than one input line. However any word that takes the next word from the input stream must find that word on the same input line. All Forth-83 standard I/O words are available as well as I$xxxx OS-9 system calls.

BUFFER SIZES

PAD is 84 bytes long, **TIB** is 128 bytes.

ERROR ACTION TAKEN BY SYSTEM

1. **Input stream exhausted before encountering a required <name> or delimiting character.**

   If the <name> is being read by **CREATE** the abort message **"--creating null word"** will be displayed. If a search is being made for a delimiting character, then the remainder of the input stream is read as if the delimiter was at the end of the stream.

2. **Insufficient stack space or insufficient number of stack entries during text interpretation or compilation.**

   If you are lucky you will get an overflow or underflow error message and abort. If not you may crash. (The underflowed word is processed before the error is detected).

3. **A word not found and not a valid number, during text interpretation or compilation;**

   The abort message **"--not in dictionary"** is displayed and **ABORT** is executed.

4. **Compilation of incorrectly nested control structures;**

   A message is printed indicating such and **ABORT** is executed.

5. **Execution of words restricted to compilation only, when not in the compile state and while not compiling a colon definition;**

   The message "--compilation only" is displayed and ABORT is executed.

6. **FORGETing within the system to a point that removes a word required for correct execution;**

   You will hang up or crash. Sorry!

7. **Insufficient space remaining in the dictionary;**

   In most cases a message will be displayed. If the word definition the caused out of memory condition was especially large, then something might have been clobbered before you see the message.

8. **A stack parameter out of range, e.g. a negative number when a +n was specified in the glossary;**

   Action may be taken anyway, in which case something could be clobbered.

9. **Correct mass storage read or write was not possible.**

   A message is displayed and ABORT will be executed. (Most common case is reading past end of file with **LIST** or **INDEX**).

Other:

Divide by zero does not cause any error action. A dividend of -1 is returned.

FLOAD

FLOAD is the loader program for the saved FORTH09 system. FLOAD loads and runs the supplied *forth* module that contains the FORTH09 core, or any Forth module saved with SAVESYS. The FORTH09 system runs in the process data area of the FLOAD process.

Syntax: FLOAD [-opts] pathname [block_path] [-opts]

The options have the form: -An

Where the "A" is any of the recognized option letters as shown below and the "n" is a decimal number. If "n" is omitted, zero is used.

OPTION	ACTION
-Bn	Allocate n block Buffers, if n is less than 2, then 2 is used. The minimum of 2 is required for EXCHANGE to function.
-Dn	Allocate a minimum of n bytes of FORTH09 Data memory. This is the memory area used for VARIABLE storage in the FORTH09 system. Any leftover memory after the other allocations will be added to the data memory allocation.
-Pn	Allocate n bytes for the PRIMARY storage area. (Includes both PRIMARY code and dictionary space.)
-Rn	Allocate n bytes for the Return stack. (The default is 256 bytes).
-Sn	Allocate n bytes for the SECONDARY storage area. (Includes both SECONDARY code and dictionary space).
-Un	Allocate n bytes for the FORTH09 data stack. (The default is 256 bytes).

None of the options for FLOAD add to the minimum process memory allocation as set up by shell. To do this you need to use the shell # parameter, and then divide the memory up as you see fit with the options.

E.g. fload Forth -b4p12000 #48k

Allocates four block buffers and 12,000 bytes of primary storage. Secondary and stack allocations will be their default values and the remainder of the 48K will be data area.

NOTE: The options can also be specified separately,
i.e. -b4 -p12000

The pathname to be loaded by FLOAD is assumed to be in the current data directory unless an explicit pathname (beginning with /) is given. The pathname module is a data module with a specific structure as setup by the word **SAVESYS** . The "block_path" is an optional mass storage file name to use instead of the default "blocks" file.

The FLOAD module contains the code for the words **SAVE** and **SAVESYS** as well as most of the block i/o words. These are accessed via a lbra table that starts 3 bytes after the module entry point. The FORTH09 word **FCALL** allows entry to these functions from Forth. Placing this code in FLOAD makes better use of memory allocation under OS-9 level II than if the code was in the FORTH09 code area.

When running the Forth system all of FORTH09's dictionaries, code areas, data areas, stacks, and block buffers occupy the FLOAD process data memory.

FORTH09 EDITOR

The "blocks" file with FORTH09 contains the source for several versions of the screen editor invoked by the word **EDIT**. The initial system LOAD from screen 1 does not contain any screen editor, only the primitive editing words **L** and **R** and **LIST** . After the initial LOAD you will need to use **INDEX** and **LIST** to find the load block for the version of the editor you will be using. Versions are contained for WYSE-50, WYSE-75, QUME QVT-102 terminals and Color Computer III 80 column screen. If you are using one of these you will only need to determine the initial load screen and load it to have a working screen editor. The version of the "blocks" file you receive may have other versions on it also, so check before you start working up your own editor. If you look at the screens with **INDEX** (see glossary) the first line of the screen to load will say "screen editor load" in it. This screen will load all other necessary screens when it is loaded.

For each different terminal type there is a "load screen" and two or more other screens containing the terminal specific words to move the cursor and interpret key strokes. If you are using a terminal type other than those provided for, you will have to create these terminal specific screens to get the editor up on that terminal. The quickest approach will probably be to select the terminal specific screens for the terminal that most closely resembles yours in function. Copy these and modify them for your terminal.

The WYSE-50 version supports editing the normal and shadow screens side-by-side in the 132 column screen format. This is a good starting point for other terminals using this feature. Most other 80 column terminals (adm-3 etc.) closely resemble the Qume QVT-102. The WYSE-75 is set up in ANSI mode, but the function key interpretation is more lengthy on this type unless the keys are manually reprogrammed at the terminal.

If the specific terminal you are using is not provided for, you will be left with only the crude line editing words L and R to edit with until you can specialize the editor for your terminal. This should be your first task. See: LIST L R in the glossary section of this manual.

The screen editor provides the following features:

- Type over editing (WYSIWIG)
- Cursor movement with cursor direction keys and tab
- Inserting blank lines or characters
- Deleting lines or characters
- Non destructive tab
- backspace to delete character before cursor
- blanking current line
- switching between editing normal screen and its shadow
- moving to previous or next screen

These features will be self explanatory as soon as you use the editor. What you see on the screen will be what is stored in the block. (WYSIWYG)

TERMINAL SPECIFIC SCREEN EDITOR WORDS

In order to specialize the screen editor for a terminal other than those provided for, use the terminal specific code for the Wyse-50 terminal as a guide line. You will need to set up a "load block", one or more blocks containing the cursor control words, and one or more blocks containing the keystroke interpretation words.

The "load block" must contain the constant definition of **WIDE?** . If the terminal you are working with has a 132 column display mode, define **WIDE?** as **TRUE** (if you want shadow screens displayed side by side with the normal screen in this mode), otherwise define WIDE? = FALSE. This should be defined as a SECONDARY word in the FORTH vocabulary. After this will be a series of **LOAD** statements (or **+LOAD**) to load the common portions of the editor and the terminal specific portions you will be defining. Fill these LOAD screen numbers in after you complete the terminal specific words.

The "cursor control" words usually fit in a single screen, if they take more, continue with a **-->** . The words that must be defined here are:

CUP	row col --	positions the cursor at row col (0,0 is upper left corner of screen)
CLS	--	clear the screen
CUU	--	move the cursor up one line
CUF	--	move the cursor forward (right) one character
CUD	--	move the cursor down one line
CUB	--	move the cursor back (left) one character
132COLS	--	switch the screen to 132 column display mode (a programmed delay may be necessary after switching modes.) If WIDE? = FALSE then make this a dummy definition.. i.e. : 132COLS ;
80COLS	--	switch the screen to 80 column display mode (a programmed delay may be necessary after switching modes.) If WIDE? = FALSE then make this a dummy definition.. i.e. : 80COLS ;
LKEY	--	display a label for the editing function keys on the bottom line(s) of the screen.
EL	--	Erase the lines displayed by LKEY

To facilitate the definition of these words any other words may be defined here. E.g. **ESC.** to send an escape character and a value , or **ESC[** to send the escape and [characters for the ANSI control sequence.

The key interpretation words usually fit in a single block, if more are necessary continue with the word **-->** . The only word that must be defined is **FINDCH** . FINDCH reads a key from the keyboard, decides what action to take, and executes the appropriate word for that action. If the editing is to terminate, FINDCH leaves a flag with the value **FALSE** on the stack, otherwise it leaves a **TRUE** flag.

There will typically be other words defined with FINDCH to do the key interpretation job, e.g. **FKEY** to interpret function key strokes, and **ESCKEY** to interpret keys that return an <ESCAPE> and other characters. The number and form of other words depends on what keys you decide to use for editing functions, and what character sequences they generate when pressed. The simplest case is where all special keys return only a single byte (as for the COCO-3 editor, look at the COCO-3 key interpretation screen).

FKEY for the WYSE-50 interprets the function keys f1..f4 which are returning the default sequence: 01 40 0D , etc. Since this is a three character sequence the other two characters must be read from the keyboard after the 01 is decoded. At this point, only the second character in the sequence is needed to determine which function key was pushed, so the other two may be dropped. The **MAYBE** statement is used to select which word to execute for each function key.

ESCKEY for the WYSE-50 interprets the edit keys to insert/delete characters and lines. Each of these keys returns an escape followed by one more character. The escape is dropped and the second character read. A **MAYBE** statement is used to select the appropriate word for each edit function.

The edit words that need to be matched to keys are given below with their function:

WORD:	FUNCTION:
IC	insert a blank character at cursor position
DC	delete character at cursor (shift rest of line left)
IL	insert a blank line at current cursor line
DL	delete current line
GOL	cursor left (GO Left)
GOR	cursor right
GOU	cursor up
GOD	cursor down
NEWL	go to next line (carriage return)
TAB	non destructive tab (tab stops every 4th character)
BSP	destructive backspace (usually matched to DEL key)
^X	erase current line (usually control/x)
<>SHADOW	toggle between shadow screen and normal screen
+SCR	edit next screen
-SCR	edit previous screen
ENDED	exit editor

Printable characters in the range 20..7E hex are placed on the stack and the word **STCH** executed to store the character in the screen buffer. Any other characters not decoded as editing functions are ignored (**DROP**ed).

EDITOR USAGE

When you exit the editor, the buffer containing the screen you just edited is marked by the word **UPDATE** to be saved later when the buffer is needed. If you exit to the operating system immediately after an edit, the buffer will NOT be written back to the block file unless you execute the word **SAVE-BUFFERS** or **FLUSH** before going to the system. It may be a good idea to execute one of these after each **EDIT**, just in case you cause your system to crash. This will ensure that the file is updated.

FORTH09 ASSEMBLER

FORTH09 contains a complete 6809 assembler under the ASSEMBLER vocabulary. All standard assembly mnemonics are supported with the exception of the branch and branch subroutine instructions. Branches are assembled by using the same conditional constructs as in the higher level FORTH words. (i.e. **IF ELSE ENDIF BEGIN UNTIL WHILE REPEAT**). A branch subroutine can be effected by switching to the normal compile state and referencing the word which will cause a bsr/lbsr to that word to be compiled.

Since all words in FORTH09 are actually compiled to machine language, (hi level words consist mostly of bsr/lbsr calls to other words), it is an easy matter to switch back and Forth between high level and assembler within a word definition. The words **[A** and **F]** switch to ASSEMBLER and back to FORTH respectively.

The assembler in FORTH09 works in reverse notation rather than the notation used by the OS-9 system assemblers *ASM* and *RMA* (or *C.ASM*). All operands must be listed before the mnemonics. When you start a code definition with the word **CODE** , the variable **STATE** remains set to execution. The assembler vocabulary words are executed to compile the machine code. This allows you to reference high level words to develop the operands (you may use arithmetic operators etc.). The mnemonics used and the instruction mode modifiers are the same as in the familiar 6809 assemblers, their order of usage to create a machine instruction is for the most part however reversed.

Any **CODE** word must preserve the values of the U and Y registers (as well as leave the S stack balanced). These may be used temporarily in the definition if they are restored upon exiting. The U register is the data stack pointer for FORTH09 and the Y register points to the base address of the data area. The value contained in the Y register is also stored at direct page address 0, so to preserve Y you have the option of either **PSH**ing and **PUL**ing it from the S stack or just trashing it and doing 0 LDY to load it from the direct page. If the CODE word definition is to use any parameters from the data stack it should of course reference these by indexing from the U register, or using PSHU PULU.

The assembler can produce extended absolute addressed mode instruction code, but for the most part this should never be used in OS-9 except by device drivers, since it will render your code position dependent and therefore illegal as an OS-9 module.

SAMPLE INSTRUCTIONS:

HEX

0 LDY	direct mode (load y from address 0)
100 LDD	extended mode (load d from address $100)
0 ,U LDD	indexed no offset (the zero is necessary)
4 ,U LDD	indexed 5 bit offset
40 ,Y LDD	indexed 8 bit offset
1000 ,S LDX	indexed 16 bit offset
,U++ LDD	indexed with auto increment
,--S LDD	indexed with auto decrement
-4 ,PC LDD	pc relative
0 ,X [] LDA	0 offset indirect indexed
,--S [] LDU	auto decrement indirect
5 ,PC [] LDD	indirect constant offset from PC
A ,X LDD	register offset from index
B ,Y STA	register offset from index
D ,U LSR	register offset from index
D ,U [] ROL	register offset indirect indexed
ABX	inherent
INCA	inherent
SWI2	inherent

```
55 # LDA                 immediate
A B CC DP  PSHU          push the a,b,cc,dp registers
D X Y U PC  PULS         pull the d,x,y,u,pc registers
X  Y  TFR                transfer the X register to Y
A  B  EXG                exchange the A and B registers
```

NOTE: In the above sample instruction list the symbol [] might appear to be a square box because of the character spacing. It is actually the left and right square bracket characters together ([ ]) and is used to denote indirect addressing.

When working in HEX keep in mind that **A B CC** and **D** are defined as registers, so for a hex literal value of for instance $D use a 0 prefix, i.e. 0D .

As stated before, the FORTH09 assembler does not include branch instructions. This is because of the need of developing the branch address without labels. The branch instructions are generated by the conditional statements, IF ENDIF, BEGIN UNTIL , etc. as in Forth. The ASSEMBLER vocabulary contains different versions of these from the FORTH vocabulary. Since the ASSEMBLER vocabulary will be the first in the search order when compiling a CODE word, the ASSEMBLER version of the conditionals will be found first in the search and used. Let's compare the use of conditional branching in the normal assembler and FORTH09 ASSEMBLER:

In normal Motorola assembler notation:

```
START    LDB #5
THERE    LSR ,S
         ROR 1,S
         DECB
         BNE THERE
```

In FORTH09 ASSEMBLER:

```
5 # LDB
BEGIN
0 ,S LSR
1 ,S ROR
DECB
0=
UNTIL
```

The second code example produces identical object code to the first. Notice in the first example that the BNE instruction branches as long as the flags indicate that the B register is "Not Equal" to zero. In the second example " 0= UNTIL " indicates that the branch is to be taken until the B register equals zero, i.e. branch whenever B <> zero. The condition you indicate in the FORTH09 assembler will always be the opposite to what the branch opcode mnemonic indicates. I.E. 0<> will generate beq, 0< will generate bpl, 0>= will generate bmi, etc. The synonyms .CS. for carry set, and .CC. for carry clear are provided for U< and U>= respectively. If you find you have difficulty getting the branch conditions correct, you may want to define similar synonyms for other conditions.

Refer to the words: **ASSEMBLER ;CODE ;CODES CODE END-CODE [A** and **F]** in the glossary.

GLOSSARY NOTATION

Portions of the glossary are copied from the FORTH-83 STANDARD, with permission and are Copyright 1983 by Forth Standards Team. (Specifically the content of standard word definitions and this preceding section).

ORDER

The glossary definitions are listed in ASCII alphabetical order.

Stack Notation

The stack parameters input to and output from a definition are described using the notation:

before -- after

before	stack parameters before execution
after	stack parameters after execution

In this notation the top of the stack is to the right. Words may also be shown in context when appropriate.

Unless otherwise noted, all stack notation describes execution time. If it applies at compile time, the line is followed by: (compiling) .

Attributes

Capitalized symbols after the stack notation indicate attributes of the defined words:

83R	The word is part of the Forth-83 **R**equired word set
83D	The word is part of the Forth-83 **D**ouble Number Extension word set
83A	The word is part of the Forth-83 **A**ssembler extension word set
83C	The word is part of the Forth-83 **C**ontrolled reference word set
83S	The word is part of the Forth-83 **S**ystem Extension word set
83U	The word is part of the Forth-83 **U**ncontrolled reference word set
83E	The word is part of the Forth-83 **E**xperimental Proposals
S	The word is a **S**ECONDARY word.
C	The word may only be used during the **C**ompilation of a colon definition.
I	The word is **I**MMEDIATE and will execute during compilation, unless special action is taken.
E	The word may be **E**xecuted only (not compiled).

Pronunciation

The natural language pronunciation of word names is given in double quotes (") where it differs from English Pronunciation.

STACK PARAMETERS

Unless otherwise stated, all references to numbers apply to 16-bit signed integers. The implied range of values is shown as {from..to}. The content of an address is shown by double braces, particularly for the contents of variables, i.e. BASE {{2..72}}.

The following are the stack parameter abbreviations and types of numbers used throughout the glossary. These abbreviations may be suffixed with a digit to differentiate multiple parameters of the same type.

Stack. Abbrv.	Number Type	Range in Decimal	Minimum Field
flag	boolean	0=false, else=true	16
true	boolean	-1 (as a result)	16
false	boolean	0	16
b	bit	{0..1}	1
char	character	{0..127}	7
8b	8 arbitrary bits (byte)	not applicable	8
16b	16 arbitrary bits	not applicable	16
n	number (weighted bits)	{-32,768..32767}	16
+n	positive number	{0..32,767}	16
u	unsigned number	{0..65,536}	16
w	unspecified weighted number (n or u)	{-32,768..65,535}	16
addr	address (same as u)	{0..65,535}	16
32b	32 arbitrary bits	not applicable	32
d	double number	{-2,147,483,648.. 2,147,483,647}	32
+d	positive double number	{0..2,147,483,647}	32
ud	unsigned double number	{0..4,294,967,295}	32
wd	unspecified weighted double number (d or ud)	{-2,147,483,648.. 4,294,967,295}	32
sys	0, 1, or more system dependent stack entries	not applicable	na

Any other symbol refers to an arbitrary signed 16-bit integer in the range {-32,768..32,767}, unless otherwise noted.

Because of the use of two's complement arithmetic, the signed 16-bit number (n) -1 has the same bit representation as the unsigned number (u) 65,535. Both of these numbers are within the set of unspecified weighted numbers (w).

INPUT TEXT

<name>

An arbitrary FORTH word accepted from the input stream. This notation refers to text from the input stream, not to values on the data stack.

ccc

A sequence of arbitrary characters accepted from the input stream until the first occurrence of the specified delimiting character. The delimiter is accepted from the input stream, but is not one of the characters ccc and is therefore not otherwise processed. This notation refers to text from the input stream, not to values on the data stack. Unless otherwise noted, the number of characters accepted may be from 0 to 255.

! 16b addr -- 83R "store"
16b is stored at addr.

!V 16b -- S,C,I "store-v"
Used in the form: 16b !V <name>
<name> can be the name of a **PRIMARY VAR** or **VARIABLE** . !V compiles fast inline code to remove 16b from the stack and store it into the VAR or VARIABLE data memory. If the variable data memory is in the direct page (i.e. doff < 100 hex when the variable was defined) then the code will be even more compact and faster.

" -- addr +n S,I "quote"
-- (compiling)
Used in the form " ccc" when compiling, a string literal ccc will be compiled such that during execution the address addr and string length +n will be left on the stack. If in execution mode, the string is moved to **PAD**.

+d1 -- +d2 83R "sharp"
The remainder of +d1 divided by the value of **BASE** is converted to an ASCII character and appended to the output string toward lower memory addresses. +d2 is the quotient and is maintained for further processing. Typically used between <# and #> .

#0 -- d "double-zero"
A double number constant, d is zero.

#> 32b -- addr +n 83R "sharp-greater"
Pictured numeric output conversion is ended dropping 32b. addr is the address of the resulting output string. +n is the number of characters in the output string. addr and +n together are suitable for **TYPE**.

#S +d -- 0 0 83R "sharp-s"
+d is converted appending each resultant character into the pictured numeric output string until the quotient (see: #) is zero. A single zero is added to the output string if the number was initially zero. Typically used between <# and #> .

#TIB -- addr 83R,S "number-t-i-b"
The address of a variable containing the number of bytes in the text input buffer. **#TIB** is accessed by **WORD** when **BLK** is zero.

$ -- S interpret only
Used to invoke an OS-9 shell command from within the Forth execution environment. The remainder of the input line after the space following $ is the shell line. E.g. $ dir

' -- addr 83R,S "tick"
Used in the form: ' <name>
addr is the compilation address of <name>. An error condition exists if <name> is not found in the currently active search order.

'FREE -- addr "tick-free"
addr is the address of the first unused byte of process DATA memory. See: **MEMLIMIT**

(-- 83R,S,I "paren"
-- (compiling)
Used in the form: (ccc)

The characters ccc, delimited by) (closing parenthesis), are considered comments. Comments are not otherwise processed. The blank following (is not part of ccc. (may be freely used while interpreting or compiling. The number of characters in ccc may be from zero to the number of characters remaining in the input stream up to the closing parenthesis.

(.) n -- "sub-dot"
Displays a number in the current number **BASE** in the manner of . <dot>, but without the trailing space.

(;) -- S "sub-semi"
Updates the permanent dictionary head pointer and code and data allocation pointers, copying the temporary working values to the permanent pointers. Used by ; at the end of a word definition when the definition is completed correctly. (;) makes the dictionary name entry made by **CREATE** appear in the dictionary and makes permanent memory allocated with **ALLOT** or **ALLOT_D** .

(COMPILE) addr -- S "sub-compile"
Compiles the word whose header address "addr" is on the stack. In FORTH09 the compilation is accomplished by compiling a bsr or lbsr instruction to the word's code field.

(CREATE) -- S,C "sub-create"
When executed will start a word definition by making a dictionary entry for the next word in the input stream. In contrast to **CREATE** , words started by **(CREATE)** do not push their parameter field address on the stack at runtime. **(CREATE)** does not affect the current value of **STATE** . The action of : <colon> is to execute **(CREATE)** and change the current **STATE** to compilation.

(FORGET) addr -- S "sub-forget"
addr is the address of a counted string (the byte at addr is the character count, and addr+1 is the first string byte) of a word name to forget in the manner of **FORGET** . See: **FORGET**

* w1 w2 -- w3 83R "times"
w3 is the least-significant 16 bits of the arithmetic product of w1 times w2.

*/ n1 n2 n3 -- n4 83R "times-divide"
n1 is first multiplied by n2 producing an intermediate 32-bit result. n4 is the floor of the quotient of the intermediate 32-bit result divided by the divisor n3. The product of n1 times n2 is maintained as an intermediate 32-bit result for greater precision than the otherwise equivalent sequence: n1 n2 * n3 / . An error condition results if the divisor is zero or if the quotient falls outside of the range {-32,768 .. 32,767}.

***/MOD** n1 n2 n3 -- n4 n5 83R "times-divide-mod"
n1 is first multiplied by n2 producing an intermediate 32-bit result. n4 is the remainder and n5 is the floor of the quotient of the intermediate 32-bit result divided by the divisor n3. A 32-bit intermediate product is used as for */ . n4 has the same sign as n3 or is zero. An error condition results if the divisor is zero or if the quotient falls outside of the range {-32,768..32,767}

+ w1 w2 -- w3 83R "plus"
w3 is the arithmetic sum of w1 plus w2.

+! w1 addr -- 83R "plus-store"
w1 is added to the w value at addr using the convention for + . This sum replaces the original value at addr.

++ -- S,C,I "plus-plus"
Used in the form: ++ <name>
<name> can be the name of a **PRIMARY VAR** or **VARIABLE** . ++ compiles fast inline code to increment the value of the variable by one. If the variable occupies the direct page the code will be more compact and faster.

+- u n1 -- n "plus-minus"
The sign of n1 is applied to u, leaving n. I.E. if n1 is negative then u is negated.

+= 16b -- S,C,I "plus-equal"
Used in the form 16b += <name>
<name> is the name of a **PRIMARY VAR** or **VARIABLE** . Compiles inline code which at runtime will remove 16b from the stack and add it to the value of <name> storing the result at <name> . The most efficient code is produced when <name> occupies direct page storage.

+BLK n1 -- n2 S "plus-b-l-k"
n2 is the sum of n1 plus the value of the variable **BLK** .

+LOAD n -- S "plus-load"
Interpretation is continued at the current **BLK** number plus n .

+LOOP n -- 83R,S,C,I "plus-loop"
sys -- (compiling)
n is added to the loop index. If the new index was incremented across the boundary between limit-1 and limit then the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding **DO** . sys is balanced with its corresponding **DO** . See: **DO**

+THRU n1 n2 -- S "plus-thru"
The consecutive blocks beginning with the current value of **BLK** plus n1 through the current value of **BLK** plus n2 are loaded.

, 16b -- 83R,S "comma"
ALLOT space for 16b then store 16b at HERE 2- .

- w1 w2 -- w3 83R "minus"
w3 is the result of subtracting w2 from w1.

-- -- S,C,I "minus-minus"
Used in the form: -- <name>
<name> is the name of a **PRIMARY VAR** or **VARIABLE** . Code is compiled to decrement the value of <name> by one. If <name> occupies the direct page the code will be shorter and faster.

--> -- 83C,S,I "next-block"
-- (compiling)
Continue interpretation on the next sequential block. May be used within a colon definition that crosses a block boundary.

-ROT 16b1 16b2 16b3 -- 16b3 16b1 16b2 "minus-rote"
The top three sixteen bit stack elements are rotated in the reverse direction from ROT, i.e. the top element is moved below the next two.

-TRAILING addr +n1 -- addr +n2 83R "dash-trailing"
The character count n1 of a text string beginning at addr is adjusted to exclude trailing spaces. If +n1 is zero, then +n2 is also zero. If the entire string consists of spaces, then +n2 is zero.

. n -- 83R "dot"
The absolute value of n is displayed in a free field format with a leading minus sign if n is negative. (A single space is displayed after the number).

." -- 83R,S,I,C "dot-quote"
-- (compiling)
Used in the form:
." ccc"
Later execution will display the characters ccc up to but not including the delimiting " (close-quote). The blank following **."** is not part of ccc.

.(-- 83R,S,I "dot-paren"
-- (compiling)
Used in the form:
.(ccc)
The characters ccc up to but not including the delimiting) (closing parenthesis) are displayed. The blank following .(is not part of ccc.

.. addr -- S "dot-dot"
Sixteen bytes starting at addr are displayed as hex numbers.

... addr -- S "dot-dot-dot"
48 bytes starting at addr are displayed as hex numbers, (3 lines of 16 bytes).

.BASE -- S "dot-base"
The current value of the variable **BASE** is displayed in decimal.

.R n +n -- 83C "dot-r"
n is converted using **BASE** and then displayed right aligned in a field +n characters wide. A leading minus sign is displayed if n is negative. If the number of characters required to display n is greater than +n an error condition exists.

.RS -- S "dot-r-s"
The elements on the *return stack* are displayed i the current number base, as unsigned numbers. The display is in order with the deepest element displayed first. The last three items on the return stack are not displayed (these are return address for the interpreter to get to the code of .RS). If nothing is on the return stack (other than the minimum three system addresses) then the message "empty" is displayed. The contents of the return stack are left unchanged. This word is useful for debugging after a word "BREAK" .

.S n1 .. nx -- n1 .. nx S,I "dot-s"
The elements on the stack n1 through nx are displayed in the current number base. The display is in order, i.e. the deepest number on the stack is displayed first. The contents of the stack are left unchanged. If the stack is empty then "empty" is displayed. See: S.

/ n1 n2 -- n3 83R "divide"
n3 is the floor of the quotient of n1 divided by the divisor n2. An error condition results if the divisor is zero or if the quotient falls outside of the range {-32,768..32,767}.

/MOD n1 n2 -- n3 n4 83R "divide-mod"
n3 is the remainder and n4 the floor of the quotient of n1 divided by the divisor n2. An error condition results if the divisor is zero or if the quotient falls outside of the range {-32,768..32,767}.

0 -- 0 "zero"
A 16b zero is left on the stack. (This word is implemented more efficiently than a Constant 0 would be.

0" -- addr S "zero-quote"
Used in the form:
0" ccc"
addr is the address of the null terminated string ccc. If compiling, a literal null terminated string is compiled such that at execution the address of this string will be left on the stack. If used in execution mode, the string will be placed in **PAD**.

0< n -- flag 83R "zero-less"
flag is true if n is less than zero (negative).

0= w -- flag 83R "zero-equals"
flag is true if w is zero.

0> n -- flag 83R "zero-greater"
flag is true if n is greater than zero.

1+ w1 -- w2 83R "one-plus"
w2 is the result of adding one to w1 according to the operation of + .

1- w1 -- w2 83R "one-minus"
w2 is the result of subtracting one from w1 according to the operation of - .

132COL? -- flag S
A constant used as a flag to control whether **SHOW** will print shadow screen side by side with normal screens. If **132COL?** is set to **TRUE** this will happen.

2! 32b addr -- 83D "two-store"
32b is stored at addr.

2+ w1 -- w2 83R "two-plus"
w2 is the result of adding two to w1 according to the operation of + .

2- w1 -- w2 83R "two-minus"
w2 is the result of subtracting two from w1 according to the operation of - .

2/ n1 -- n2 83R "two-divide"
n2 is the result of arithmetically shifting n1 right one bit. The sign is included in the shift and remains unchanged.

2@ addr -- 32b 83D "two-fetch"
32b is the value at addr.

2* w1 -- w2 83C "two-times"
w2 is the result of shifting w1 left one bit. A zero is shifted into the vacated bit position.

2CONSTANT 32b -- 83D,S "two-constant"
A defining word executed in the form:
32b 2CONSTANT <name>
Creates a dictionary entry for <name> so that when <name> is later executed, 32b will be left on the stack.

2DROP 32b -- 83D "two-drop"
32b is removed from the stack.

2DUP 32b -- 32b 32b 83D "two-dupe"
Duplicate 32b.

2OVER 32b1 32b2 -- 32b1 32b2 32b3 83D "two-over"
32b3 is a copy of 32b1.

2ROT 32b1 32b2 32b3 -- 32b2 32b3 32b1 83D "two-rote"
The top three double numbers on the stack are rotated, bringing the third double number to the top of the stack.

2SWAP 32b1 32b2 -- 32b2 32b1 83D "two-swap"
The top two double numbers are exchanged.

2VARIABLE -- 83D,S "two-variable"
A defining word executed in the form:
2VARIABLE <name>
A dictionary entry for <name> is created and four bytes are **ALLOT**ed in its parameter field. This parameter field is to be used for contents of the variable. The application is responsible for initializing the contents of the variable which it creates. When <name> is later executed, the address of its parameter field is placed on the stack. *NOTE: In FORTH09 the variable parameter field is in process data memory.*

: -- sys 83R,S "colon"
A defining word executed in the form:
: <name> ... ;
Creates a word definition for <name> in the compilation vocabulary and sets compilation state. The search order is changed so that the first vocabulary in the search order is replaced by the compilation vocabulary. The compilation vocabulary is unchanged. The text from the input stream is subsequently compiled. <name> is called a "colon definition". The newly created word definition for <name> cannot be found in the dictionary until the corresponding ; or **;CODE** is successfully processed. An error condition exists if a word is not found and cannot be converted to a number or if, during compilation from mass storage, the input stream is exhausted before encountering ; or **;CODE** . sys is balanced with its corresponding ; .

; -- 83R,S,C,I "semi-colon"
sys -- (compiling)
Stops compilation of a colon definition, allows the <name> of this colon definition to be found in the dictionary, sets interpret state and compiles **EXIT** (or equivalent). sys is balanced with its corresponding **:** .

;CODE -- 83A,S,C,I "semi-colon-code"
sys1 -- sys2 (compiling)
Used in the form:
: <namex> ... <create> ... ;CODE ... END-CODE
Stops compilation, terminates the defining word <namex> and executes ASSEMBLER. When <namex> is executed in the form:<namex> <name> to define the new <name>, the execution address of <name> will be the address of the code sequence following the **;CODE** . Execution of any <name> will cause this machine code sequence to be executed, *this machine code sequence will reside in the primary code area in FORTH09 even though the colon definition may be for a secondary word. sys1 is balanced with its corresponding : . sys2 is balanced with its corresponding END-CODE . See: CODE ;CODES DOES> DOES>S*

;CODES -- S,C,I "semi-colon-code-s"
sys1 -- sys2 (compiling)
Used in the form:
: <namex> ... <create> ... ;CODES ... END-CODE
Functions the same as **;CODE** except the machine code defined between **;CODES** and **END-CODE** will reside in the **SECONDARY** code area. Used to make defining words that will only be used by secondary code. See: **;CODE**

;S -- S,E "semi-s"
Stop interpretation of a block

< n1 n2 -- flag 83R "less-than"
flag is true if n1 is less than n2.

<# -- 83R "less-sharp"
Initialize pictured numeric output conversion
The words: **# #> #S <# HOLD SIGN**
can be used to specify the conversion of a double number into an ASCII text string stored in right-to-left order. NOTE: see also the non-standard word _SIGN .

<> w1 w2 -- flag 83U "not-equal"
flag is true if w1 is not equal to w2.

<MARK -- addr 83S,S,C "backward-mark"
Used at the destination of a backward branch. addr is typically only used by **<RESOLVE** to compile a branch address.

<RESOLVE addr -- 83S,S,C "backward-resolve"
Used at the source of a backward branch after either **BRANCH** or **?BRANCH** . Compiles a branch address using addr as the destination address.

= w1 w2 -- flag 83R "equals"
flag is true if w1 is equal to w2.

> n1 n2 -- flag 83R "greater-than"
flag is true if n1 is greater than n2. (signed comparison).

>< 16b1 -- 16b2 83U "byte-swap"
16b2 is the result of swapping the high and low bytes within 16b1.

>BODY addr1 -- addr2 83R,S "to-body"
addr2 is the parameter field address corresponding to the compilation address addr1.

>ERROR# n -- "to-error-number"
n is stored in the **VAR ERROR#** (which records the errors returned by OS-9 system call words).

>IN -- addr 83R,S "to-in"
The address of a variable which contains the present character offset within the input stream {{0..the number of characters in the input stream}}.

>IN' -- n S "to-in-prime"
or n --
>IN' is a **VAR** used to record the value of **>IN** before it is cleared by **QUIT** so that the location of an error may be determined.

>MARK -- addr 83S,S,C "forward-mark"
Used at the source of a forward branch. Typically used after either **BRANCH** or **?BRANCH** . Compiles space in the dictionary for a branch address which will later be resolved by **>RESOLVE** .

>PATH addr -- S "to-path"
addr is the address of an OS-9 pathname string. All output to path 1 (normally the screen) is redirected to the new path designated by **>PATH** . The new path is created using the OS-9 **I$CREATE** system call. If an error occurs when attempting the create the path, the output remains directed to the screen. See: **>PATH+ >SCREEN >PRINTER**

>PATH+ addr -- S "to-path-plus"
addr is the address of an OS-9 pathname string. All output to path 1 (normally the screen) is redirected and appended to the end of the designated path. The designated path is opened with the OS-9 **I$OPEN** system call and must be accessible otherwise the output remains directed to the screen. See: **>PATH >SCREEN >PRINTER**

>PRINTER -- S "to-printer"
The output of all Forth words which normally write to the screen (path 1) is redirected to the printer. This remains in effect until output is again redirected with one of the words: **>SCREEN >PATH >PATH+** . *NOTE: The default printer path for this word is set to "/p1", you must edit the source for this word and recompile it in order to use a different printer path.*

>R 16b -- 83R,C "to-r"
Transfers 16b to the return stack.

>RESOLVE addr -- 83S,S,C "forward-resolve"
Used at the destination of a forward branch. Calculates the branch address (to current location in the dictionary) using addr and places this branch address into the space left by **>MARK** .

>SCREEN -- S "to-screen"
Redirects the output of all Forth words that write to path 1 (the screen) back to the screen. Used after redirecting screen output with **>PRINTER >PATH** or **>PATH+** to return output to the screen.

?BRANCH flag -- 83S,S,C "question-branch"
When used in the form: COMPILE ?BRANCH a conditional branch operation is compiled. See BRANCH for further details. When executed, if flag is false the branch is performed as with **BRANCH** . When flag is true execution continues at the compilation address immediately following the branch address.

?BRANCH_ flag -- S,C
Equivalent to the sequence: COMPILE ?BRANCH . Causes the compilation of a conditional branch operation. When executed, if flag is false the branch is performed as with **BRANCH** . When flag is true execution continues at the compilation address immediately following the branch address.

?TBRANCH_ flag -- S,C
Causes the compilation of a conditional branch with the opposite polarity of **?BRANCH**. I.e. when the branch code is executed the branch will be taken if the flag is **TRUE**. When the flag is false execution will continue at the compilation address immediately following the branch address.

?DUP 16b -- 16b 16b 83R "question-dup"
or 0 -- 0
Duplicate 16b if it is non-zero.

?EXEC -- S "q-exec"
An error message is displayed and **ABORT** executed if the current state is not execution.

?LEAVE flag -- S,C "question-leave"
Causes termination of a do loop if flag is true. May only be used within a **DO** loop construct.

?PAIR u1 u2 -- S "q-pair"
Displays a message and **ABORT**s is u1 <> u2 .

@ addr -- 16b 83R "fetch"
16b is the value at addr.

@V -- 16b S,C,I "fetch-v"
Used in the form: @V <name>
<name> is the name of a PRIMARY **VAR** or **VARIABLE** . Compiles inline code to fetch the value of the variable and place it on the stack. The most efficient code results when the variable occupies the direct page.

ABORT 83R,S "abort"
Clears the data stack and performs the function of **QUIT** . No message is displayed.

ABORT" flag -- 83R,S,C,I "abort-quote"
-- (compiling)
Used in the form:
flag ABORT" ccc"
When later executed, if flag is true the characters ccc, delimited by " (close-quote) are displayed and then a system dependent error abort sequence, including the function of **ABORT** , is performed. If flag is false, the flag is dropped and execution continues. The blank following **ABORT"** is not part of ccc.

ABS n -- u 83R "absolute"
u is the absolute value of n. If n is -32,768 then u is the same value.

AGAIN -- 83U,S,C,I
sys -- (compiling)
Effect an unconditional jump back to the start of a BEGIN AGAIN loop. sys is balanced with its corresponding **BEGIN** . See; **BEGIN**

ALLOT w -- 83R,S
Allocates w bytes in the dictionary. The address of the next available dictionary location is updated accordingly. (NOTE: allot allocates space in the code area, See: **ALLOT_D**)

ALLOT_D w -- S "allot-data"
Allocates w bytes of process data space. *NOTE: In the Forth-83 standard no distinction is made between code space and data space, however this distinction is necessary for processes running under OS-9, hence the distinction here.*

ALSO -- 83E,S
The transient vocabulary becomes the first vocabulary in the resident portion of the search order. Up to the last five resident vocabularies will also be reserved, in order, forming the resident search order.

AND 16b1 16b2 -- 16b3 83R
16b3 is the bit-by-bit logical 'and' of 16b1 with 16b2.

ASCII -- +n 83U,S,I
Used in the form: ASCII c
+n is the ASCII character value of c which is the first (or only) letter of the next word in the input stream after **ASCII** .

ASSEMBLER -- 83A,S
Execution replaces the first vocabulary in the search order with the **ASSEMBLER** vocabulary. See: **VOCABULARY**

A[-- S,C,I "a-left-bracket"
Invokes the **ASSEMBLER** vocabulary without changing **STATE** . This is used to compile assembler words in a definition such that when later executed the assembly code will be compiled (at location **HERE**). The corresponding **F[** shifts back to the **FORTH** vocabulary. Contrast this with **[A** which switches to assembler in the execute mode resulting in compilation of the assembly words inline in the current definition.

BASE -- addr 83R
The address of a variable containing the current numeric conversion radix. {{2..72}}

BEGIN -- 83R,S,C,I
-- sys (compiling)
Used in the form:
BEGIN ... flag UNTIL
or
BEGIN ... flag WHILE ... REPEAT
BEGIN marks the start of a word sequence for repetitive execution. A BEGIN-UNTIL loop will be repeated until flag is true. A BEGIN-WHILE-REPEAT loop will be repeated until flag is false. The words after **UNTIL** or **REPEAT** will be executed when either loop is finished. sys is balanced with its corresponding UNTIL or WHILE .

BLANK addr u -- 83C
u bytes of memory beginning at addr are set to the ASCII character value for space. No action is taken if u is zero.

BLK -- addr 83R,S "b-l-k"
The address of a variable containing the number of the mass storage block being interpreted as the input stream. If the value of **BLK** is zero the input stream is taken from the text input buffer. {{0..the number of blocks available -1 }}

BLK' -- n S "b-l-k-prime"
n is the former value of **BLK** before it was set to zero by **ABORT** .

BLK-ERASE u -- S "b-l-k-erase"
Mass storage block u and its shadow block are erased (to all space characters).

BLKPTR -- addr S "block-pointer"
addr is the address of the oldest block buffer structure in the chain. **BLKPTR** is type **VAR**. The value at addr is the link to the next buffer, actual buffer starts at addr+2.

BLOCK u -- addr 83R,S
addr is the address of the assigned buffer of the first byte of block u. If the block occupying that buffer is not block u and has been UPDATEd it is transferred to mass storage before assigning the buffer. If block u is not already in memory, it is transferred from mass storage into an assigned block buffer. A block may not be assigned to more than one buffer. If u is not an available block number, an error condition exists. Only data within the last buffer referenced by **BLOCK** or **BUFFER** is valid. The contents of a block buffer must not be changed unless the change may be transferred to mass storage.

BRANCH -- 83S,S,C
When used in the form: COMPILE BRANCH an unconditional branch operation is compiled. A branch address must be compiled immediately following this compilation address. The branch address is typically generated by following **BRANCH** with **<RESOLVE** or **>MARK** .

BRANCH_ -- S,C
When executed, compiles an unconditional branch operation as in: COMPILE BRANCH . See: **BRANCH** for more details.

BREAK -- S

When BREAK is compiled into a word definition, and later executed, it causes the execution of that word to be suspended and the interpreter to be entered. Any words can then be executed to aid in debugging, and then the execution of the original word resumed by executing the word **GO** . If any error condition causes the words **ABORT** or **QUIT** to executed then execution of the word with the BREAK can not be resumed. See: **GO .S .RS**

BUFFER u -- addr 83R,S

Assigns a block buffer to block u. addr is the address of the first byte of the block within its buffer. This function is fully specified by the definition for **BLOCK** except that if the block is not already in memory it might not be transferred from mass storage. The contents of the block buffer assigned to block u by **BUFFER** are unspecified.

C! 16b addr -- 83R "c-store"

The least-significant 8 bits of 16b are stored into the byte at addr.

C, 16b -- 83C,S "c-comma"

ALLOT one byte then store the least-significant 8 bits of 16b at HERE 1- .

C@ addr -- 8b 83R "c-fetch"

8b is the contents of the byte at addr. NOTE: a 16 bit word is actually put on the stack with the upper 8 bits being zero.

CAPS -- flag S

CAPS is a VAR used as a flag by WORD. If caps is set to true (e.g. TRUE TO CAPS), then word translates all characters from the input stream to uppercase. The effect is the words in lowercase in the input stream will match uppercase only words in the dictionary. While CAPS is set to TRUE it is not possible to match any word in the dictionary containing one or more lowercase letters.

CMDLINE -- addr "command-line"

addr is the address of the shell command line parameters when the **PRIMARY SAVE**d program was invoked. In the development environment it is the command line when FLOAD was invoked. **CMDLINE** is type **VAR** so the value may be altered for testing.

CMOVE addr1 addr2 u -- 83R "c-move"

Move u bytes beginning at address addr1 to addr2. The byte at addr1 is moved first, proceeding toward high memory. If u is zero nothing is moved.

CMOVE> addr1 addr2 u -- 83R "c-move-up"

Move the u bytes at address addr1 to addr2. The move begins by moving the byte at (addr1 plus u minus 1) to (addr2 plus u minus 1) and proceeds to successively lower addresses for u bytes. If u is zero nothing is moved. (Useful for sliding a string towards higher addresses).

CODE -- sys 83A,S

A defining word executed in the form:

CODE <name> ... END-CODE

Creates a dictionary entry for <name> to be defined by a following sequence of assembly language words. Words thus defined are called code definitions. This newly created word definition for <name> cannot be found in the dictionary until the corresponding **END-CODE** is successfully processed (see: **END-CODE**). Executes **ASSEMBLER** . sys is balanced with its corresponding **END-CODE** .

COMP? -- S "compiling-?"
Displays an error message and **ABORT**s if not compiling.

COMPILE -- 83R,S,C
Typically used in the form:
: <name> ... COMPILE <namex> ... ;
When <name> is executed, the compilation address compiled for <namex> is compiled and not executed. <name> is typically immediate and <namex> is typically not immediate.

COMPLIMIT -- n or TO n -- S "compile-limit"
A VAR containing the maximum number of bytes to compile as inline code rather than as a subroutine call. If a **CODE** word's flag bits indicate that it may be compiled inline, and the total code bytes are less than or equal to **COMPLIMIT** when that word is compiled into a new word definition, then the code word is copied as inline code (less the rts) rather than as a bsr or lbsr to the code.

COMPONLY -- S "compilation-only"
Sets a flag bit in the dictionary header of the last defined word to indicate that the word may be compiled only, i.e. used within a colon definition. If an attempt is made to execute the word directly, an error message will be displayed and a call to **ABORT** executed.

CONSTANT 16b -- 83R,S
A defining word executed in the form:
16b CONSTANT <name>
creates a dictionary entry for <name> so that when <name> is later executed, 16b will be left on the stack.

CONTEXT -- addr 83S,S
The address of a variable which determines the dictionary search order. *In FORTH09 addr is the address of an array of 7 pointers pointing to the vocabulary tables of the vocabularies in the search order.*

CONVERT +d1 addr1 -- d2 addr2 83R
+d2 is the result of converting the characters within the text beginning at addr1+1 into digits, using the value of **BASE** , and accumulating each into +d2 after multiplying +d1 by the value of **BASE** . Conversion continues until an unconvertible character is encountered. addr2 is the location of the first unconvertible character.

CONVEY u1 u2 u3 -- S
The range of mass storage screens and their shadow screens from u1 through u2 is exchanged with the range of screens beginning with u3 . See also: **SLIDE EXCHANGE**

COPY u1 u2 -- S
The mass storage screen u1 and its shadow are copied to screen u2 (and its shadow).

COUNT addr1 -- addr2 +n 83R
addr2 is addr1+1 and +n is the length of the counted string at addr1. The byte at addr1 contains the byte count +n. Range of +n is {0..255}

CR -- 83R "c-r"
Displays a carriage-return and line-feed or equivalent operation.

CR" -- addr u S,I "c-r-quote"
-- (compiling)
Used in the form CR" ccc" . If compiling, compiles a literal string ccc terminated with an ASCII cr, such that the address addr, and byte count u, of this string is placed on the stack at runtime. The count u includes the appended CR character. If in execution mode, this string is placed in **PAD**.

CREATE -- 83R,S
A defining word executed in the form:
CREATE <name>
creates a dictionary entry for <name>. After <name> is created, the next available dictionary location is the first byte of <name>'s parameter field. When <name> is subsequently executed, the address of the first byte of <name>'s parameter field is left on the stack. **CREATE** does not allocate space in <name>'s parameter field.

CURHEAD -- addr S
addr is the address of the current dictionary head either **PRIMARY** or **SECONDARY** depending upon the value of **PSTATE**. **CURHEAD** is type **VAR** .

CURRENT -- addr 83S,S
The address of a variable specifying the vocabulary in which new word definitions are appended.

D! 32b addr -- "d-store"
32b is stored at addr. (This is a synonym for 2!).

D* wd1 wd2 -- wd3 "d-times"
wd3 is the result of multiplying wd1 by wd2.

D+ wd1 wd2 -- wd3 83R "d-plus"
wd3 is the arithmetic sum of wd1 plus wd2.

D+- ud n -- d "d-plus-minus"
d is the result of applying the sign of n to ud. I.e. if n is negative then ud is negated.

D- wd1 wd2 -- wd3 83D "d-minus"
wd3 is the result of subtracting wd2 from wd1.

D. d -- 83D "d-dot"
The absolute value of d is displayed in a free field format. A leading negative sign is displayed if d is negative. (A single space is displayed after the number).

D.R d +n -- 83D "d-dot-r"
d is converted using the value of **BASE** and then displayed right aligned in a field +n characters wide. A leading minus sign is displayed if d is negative. If the number of characters required to display d is greater than +n, an error condition exists.

D/ d1 d2 -- d3 "d-slash"
d3 is the floor of the quotient of d1 divided by d2. An error condition exists if the divisor is zero.

D0= wd -- flag 83D "d-zero-equals"
flag is true if wd is zero.

D2/ d1 -- d2 83D "d-two-divide"
d2 is the result of d1 arithmetically shifted right one bit. The sign is included in the shift and remains unchanged.

D< d1 d2 -- flag 83R "d-less-than"
flag is true if d1 is less than d2 according to the operation of < except extended to 32 bits.

D= wd1 wd2 -- flag 83D "d-equal"
flag is true if wd1 equals wd2.

D@ addr -- 32b "d-fetch"
32b is the value at addr. (This is a synonym for 2@).

DABS d -- ud 83D "d-absolute"
ud is the absolute value of d. If d is -2,147,483,648 then ud is the same value.

DCONSTANT 32b -- S "d-constant"
A defining word executed in the form:
32b DCONSTANT <name>
Creates a dictionary entry for <name> so that when <name> is later executed, 32b will be left on the stack. (This is a synonym for **2CONSTANT**).

DDROP 32b -- "d-drop"
32b is removed from the stack. (This is a synonym for **2DROP**).

DDUP 32b -- 32b 32b "d-dupe"
Duplicate 32b. (This is a synonym for **2DUP**).

DECIMAL -- 83R
Set the input-output numeric conversion base to ten.

DEFINITIONS -- 83R,S
The compilation vocabulary is changed to be the same as the first vocabulary in the search order.

DEPTH -- +n 83R
+n is the number of 16-bit values contained in the data stack before +n was placed on the stack.

DIGIT c -- n flag or
c -- flag
Attempts to convert the ASCII character c to binary using the value of BASE. If conversion is possible then n is the binary digit value and flag is true, else flag is false.

DLITERAL -- 32b S,C,I "d-literal"
32b -- (compiling)
Compiles a double literal such that when later executed 32b is left on the stack.

DMAX d1 d2 -- d3 83D "d-max"
d3 is the greater of d1 and d2.

DMIN d1 d2 -- d3 83D "d-min"
d3 is the lesser of d1 and d2.

DNEGATE d1 -- d2 83R "d-negate"
d2 is the two's complement of d1.

DO w1 w2 -- 83R,S,C,I
-- sys (compiling)
Used in the form:
DO ... LOOP
or
DO ... +LOOP
Begins a loop which terminates based on control parameters. The loop index begins at w2, and terminates based on the limit w1. See **LOOP** and **+LOOP** for details on how the loop is terminated. The loop is always executed at least once. For example: w DUP DO ... LOOP executes 65,536 times. sys is balanced with its corresponding **LOOP** or **+LOOP**. An error condition exists if insufficient space is available for at least three nesting levels.

DOES> -- addr 83R,S,C,I "does"
-- (compiling)
Defines the execution-time action of a word created by a high-level defining word. Used in the form:

: <namex> ... <create> ... DOES> ... ;

and then

<namex> <name>

where <create> is **CREATE** or any user defined word which executes **CREATE** . Marks the termination of the defining part of the defining word <namex> and then begins the definition of the execution-time action for words that will later be defined by <namex>. When <name> is later executed, the address of <name>'s parameter field is placed on the stack and then the sequence of words between DOES> and ; are executed. *FORTH09: The code between DOES> and ; will reside in the PRIMARY code area even if namex is a SECONDARY word.*

DOES>S -- addr S,C,I "does-s"
-- (compiling)
Defines the execution-time action of a word created by a high-level defining word. Used in the form:

: <namex> ... <create> ... DOES>S ... ;

and then

<namex> <name>

where <create> is **CREATE** or any user defined word which executes **CREATE** . Marks the termination of the defining part of the defining word <namex> and then begins the definition of the execution-time action for words that will later be defined by <namex>. When <name> is later executed, the address of <name>'s parameter field is placed on the stack and then the sequence of words between **DOES>** and ; are executed. *FORTH09: The code between DOES> and ; will reside in the SECONDARY code area.*

DOVER 32b1 32b2 -- 32b1 32b2 32b3 "d-over"
32b3 is a copy of 32b1. (This is a synonym for **2OVER**).

DROP 16b -- 83R
16b is removed from the stack.

DROT 32b1 32b2 32b3 -- 32b2 32b3 32b1 "d-rote"
The top three double numbers on the stack are rotated, bringing the third double number to the top of the stack. (This is a synonym for **2ROT**).

DSWAP 32b1 32b2 -- 32b2 32b1 "d-swap"
The top two double numbers are exchanged. (This is a synonym for **2SWAP**).

DU/MOD 32b1 32b2 -- 32b3 32b4 "d-u-divide-mod"
32b3 is the quotient and 32b4 is the remainder of 32b1 divided by 32b2. An error condition exists if the divisor is zero.

DU< ud1 ud2 -- flag 83D "d-u-less"
flag is true if ud1 is less than ud2. Both numbers are unsigned.

DUM* 32b1 16b -- 32b2 "dumb-star"
32b2 is the least significant 32 bits of the result of multiplying 32b1 by 16b .

DUP 16b -- 16b 16b 83R "dupe"
Duplicate 16b.

DUMP addr u -- 83C
List the contents of u addresses starting at addr. Each line of values may be preceded by the address of the first value.

DVARIABLE -- S "d-variable"
A defining word executed in the form:
DVARIABLE <name>
A dictionary entry for <name> is created and four bytes are **ALLOT**ed in its parameter field. This parameter field is to be used for contents of the variable. The application is responsible for initializing the contents of the variable which it creates. When <name> is later executed, the address of its parameter field is placed on the stack. (This is a synonym for **2VARIABLE**). *NOTE: In FORTH09 the variable parameter field is in process data memory.*

ECHK -- "e-check"
ECHK is used in OS-9 system call words immediately after the SWI2 to store the returned error code from the call to ERROR# .

EDIT u -- S
Invoke the screen editor to edit block u.

EDITION u -- S
The edition number encoded in the module header of modules later saved with **SAVE** **XSAVE** or **SAVESYS** will be set to u .

ELSE -- 83R,S,C,I
sys1 -- sys2 (compiling)
Used in the form:
flag IF ... ELSE ... THEN
ELSE executes after the true part following **IF** . **ELSE** forces execution to continue at just after **THEN** (or **ENDIF**) . sys1 is balanced with its corresponding IF . sys2 is balanced with its corresponding **THEN** . See: **IF THEN** (and also **ENDIF**)

EMIT 16b -- 83R

The least-significant 7-bit ASCII character is displayed. *NOTE: In FORTH09 the least significant 8 bits are transmitted.*

EMPTY-BUFFERS -- 83C,S "empty-buffers"

Unassign all block buffers. **UPDATE**d blocks are not written to mass storage.

END-CODE sys -- 83A,S "end-code"

Terminates a code definition and allows the <name> of the corresponding code definition to be found in the dictionary. sys is balanced with its corresponding CODE or ;CODE . See: CODE

ENDIF -- S,C,I

sys -- (compiling)

Used in the form:

flag IF ... ELSE ... ENDIF

or

flag IF ... ENDIF

ENDIF is the point where execution continues after **ELSE** , or **IF** when no **ELSE** is present. sys is balanced with its corresponding **IF** or **ELSE** . See: **IF ELSE** . *NOTE: ENDIF is a synonym for THEN but is not a part of the FORTH-83 Standard.*

EOF? u -- flag "e-o-f"

flag will be true if OS-9 path number u is currently positioned at end of file. An error condition exists if u is not a valid open path number, in which case the **ERROR#** will be non-zero.

EQU -- <name1> <name2> S "equate"

A new entry is created in the dictionary for <name1> and its code field and permission is made the same as <name2>. EQU makes an alias name for an existing word definition, both the new and old names point to the same code.

ERASE addr u -- 83C

u bytes of memory beginning at addr are set to zero. No action is taken if u is zero.

ERROR# -- n "error-number"

n is the error number returned by the last OS-9 system call word. If ERROR# equals zero then no error occurred. Range of n is {0..255}.

EXCHANGE u1 u2 -- S

Mass storage screen u1 is exchange with screen u2, i.e. after execution of exchange, u2 will reside in the screen formerly occupied by u1 and visa versa.

EXECUTE addr -- 83R

The word definition indicated by addr is executed. An error condition exists if addr is not a compilation address.

EXIT -- 83R,C

Compiled within q colon definition such that when executed, that colon definition returns control to the definition that passed control to it by returning control to the return point on the top of the return stack. An error condition exists if the top of the return stack does not contain a valid return point. May not be used within a do-loop.

EXPECT addr +n -- 83R

Receive characters and store each into memory. The transfer begins at addr proceeding towards higher addresses one byte per character until either a "return" is received or until +n characters have been transferred. No more than +n characters will be stored. The "return" is not stored into memory. No characters are received or transferred if +n is zero. All characters actually received and stored into memory will be displayed, with the "return" displaying as a space. See: SPAN

F$ALLRAM u1 -- u2 LEVEL II OS-9 ONLY

#of_blks -- starting_blk#

Request u1 memory blocks be allocated. u2 is the block number of the first block allocated. Block size is system dependent on the MMU. Blocks allocated are from unallocated memory pool outside of the process memory. The requested blocks are NOT mapped into the process that issues this request, use **F$MAPBLK** to do this. See: **F$DELRAM F$MAPBLK F$CLRBLK**

F$CHAIN addr1 u1 addr2 u2 u3 -- "f-dollar-sign-chain"

param_addr param_size name_addr data_size lang_type

addr1 is the parameter field address, u1 is the parameter field size, addr2 is the name string address, u2 is the optional data size {0..255}, and u3 is the language-type byte for an OS-9 F$CHAIN system call. See the OS-9 technical manual for further details.

F$CLRBLK addr u -- LEVEL II OS-9 ONLY

(addr_of_first #of_blks --)

Unmaps u blocks of ram beginning at addr (within the process's memory map) from the process's memory map. This is the inverse of the **F$MAPBLK** function. addr should be on a block boundary. Block size is system dependent on the system MMU. See: **F$MAPBLK F$ALLRAM F$DELRAM**

F$CRC addr1 u addr2 -- "f-dollar-sign-c-r-c"

(crc_accum byte_cnt beg_addr)

addr1 is the address of the 3 byte crc accumulator, u is the byte_count, and addr2 is the starting address for an OS-9 F$CRC system call.

F$DELRAM u1 u2 -- LEVEL II OS-9 ONLY

(starting_blk# #of_blks --)

Deallocates u2 blocks of memory beginning with block number u1. The blocks deallocated return to the OS-9 free memory pool. Normally used to deallocate blocks that were allocated by the **F#ALLRAM** call.

F$FORK addr1 u1 addr2 u2 u3 -- addr3 u4

(param_addr param_sz name_addr data_sz lang_type --
name_addr+ proc_ID)

addr1 is the parameter field address, u1 is the parameter field size, addr2 is the module name string address, u2 is the optional data_size and u3 is the language/type byte for an OS-9 F$FORK system call. On return addr3 is the module name ptr (addr2) incremented past the name and u4 is the new process ID.

F$ID -- u1 u2

(-- user_ID process_ID)

u1 is the user_ID and u2 is the process_ID returned by an OS-9 F$ID system call.

F$LINK addr1 u1 -- addr2 addr3 addr4 u2 u3
(name_ptr type_lang --
mod_addr entry_addr name_ptr+ attr/rev type/lang)
addr1 is the module name string address, and u1 is the type/language byte of the module to be linked by an OS-9 F$LINK call. On return addr2 will be the modules header address, addr3 the entry address, addr4 will point past the name string, u2 the linked module attribute/revision byte, and u3 the module type_language byte.

F$LOAD addr1 u1 -- addr2 addr3 addr4 u2 u3
(name_ptr type/lang --
mod_addr entry_addr name_ptr+ att/rev typ/lang)
addr is the module name string address, and u1 is the type/language byte of the module to be loaded by an OS-9 F$LOAD call. On return addr2 will be the module's header address, addr3 the module entry point, addr4 will point past the name string, u2 is the attribute/revision byte, and u3 is the type/language byte.

F$MAPBLK u1 u2 -- addr LEVEL II OS-9 ONLY
(starting_blk# #of_blks -- addr_of_first)
Maps u2 blocks of memory, beginning with block number u1 into the process's memory map. addr is the address of the first block within the process's memory map. This address would be on a block boundary. Block size is dependent on the system MMU (usually 2K, 4K, or 8K). See: **F$ALLRAM F$DELRAM F$CLRBLK**

F$MEM u1 -- addr u2
(desired_mem_size -- upper_bound actual_size)
u1 is the new desired data memory size (0 returns actual memory size). addr is the address of the last byte of the data memory and u2 is the actual new memory size.

F$PERR u --
u is the error code to be printed by the OS-9 F$PERR system call.

F$SEND u1 u2 --
(signal proc_id --)
u1 is the signal code that will be sent to process u2 by the OS-9 F$SEND call.

F$SLEEP u1 -- u2
(tick_cnt -- updated_tick_cnt)
u1 is the tick count for an F$SLEEP system call, u2 will be the updated tick count on return from the call.

F$TIME addr --
addr is the address of a 6 byte data area where the time packet will be returned by the OS-9 F$TIME call.

F$UNLINK addr --
addr is the module header address of the module to be unlinked.

F$UNLOAD addr u -- Level II OS-9 ONLY
(name_ptr type/lang --)
addr is the module name string pointer and u is the type/language byte of the module to unload. NOTE: F$UNLOAD is a level 2 OS-9 call only.

F$WAIT -- u1 u2
(-- child's_status child's_ID)
u1 is the child's status code and u2 is the child's ID code. NOTE: F$WAIT is only used after a F$FORK call.

FALSE -- 0
A constant with a value of 0 (i.e. flag false value).

FCALL 8b -- S
A subroutine call is made into the FLOAD module to a jump table offset of 8b.

FCS" -- addr S,C,I "f-c-s-quote"
Used in the form: FCS" ccc"
generates a string literal of ccc with the most significant bit of the last character of the string set. At runtime addr will be the address of this string. NOTE: This is useful for various OS-9 system calls that use hi bit terminated strings.

FILELOAD -- nnn S
A **LOAD** operation is done from the OS-9 pathname nnn. If nnn is not found the message "file error" will be displayed and **ABORT** executed.

FILL addr u 8b -- 83R
u bytes of memory beginning at addr are set to 8b. No action is taken if u is zero.

FIND addr1 -- addr2 n 83R,S
addr1 is the address of a counted string. The string contains a word name to be located in the currently active search order. If the word is not found, addr2 is the string address addr1, and n is zero. If the word is found, addr2 is the compilation address and n is set to one of two non-zero values. If the word found has the immediate attribute, n is set to one. If the word is non-immediate, n is set to minus one (true).

FLUSH -- 83R,S
Performs the function of **SAVE-BUFFERS** then unassigns all block buffers. (This may be useful for mounting or changing mass storage media).

FORGET -- 83R,S
Used in the form:
FORGET <name>
If <name> is found in the compilation vocabulary, delete <name> from the dictionary and all words added to the dictionary after <name> regardless of their vocabulary. Failure to find <name> is an error condition. An error condition also exists if the compilation vocabulary is deleted.

FORTH -- 83R,S
The name of the primary vocabulary. Execution replaces the first vocabulary in the search order with **FORTH** . **FORTH** is initially the compilation vocabulary and the first vocabulary in the search order. New definitions become part of the **FORTH** vocabulary until a different compilation vocabulary is established. See: **VOCABULARY**

FORTH-83 -- 83R,S
Assures that a FORTH-83 Standard System is available, otherwise an error condition exists. *NOTE: In FORTH09 this word doesn't do much except set the initial vocabulary to FORTH and LOAD from block 1 if the word 2! is not found in the dictionary. If the mass storage file you are USING is not the original blocks file then you may NOT actually have a FORTH-83 standard system after executing FORTH-83.*

F[-- S,C,I "f-left-bracket"
Executes **FORTH**, making it the first vocabulary in the search order. Usually used in conjunction with **A[** . See: **A[**

F] -- S "f-right-bracket"
Resume compilation and execute the **FORTH** vocabulary.

GET.DEVNM addr u -- "get-dev-name"
(buffer_addr path# --)
The device name of open path number u is copied into the 32 byte buffer at addr. NOTE: The device name is a high bit terminated string.

GET.OPT addr u -- "get-options"
(buffer_addr path# --)
The device options of path u are copied to the 32 byte buffer at addr.

GET.POS u -- ud "get-position"
(path# -- file_position)
ud is the current file position of path u.

GET.SIZE u -- ud
(path# -- file_size)
ud is the current file size of path u

GETCH u -- c
(path# -- char)
c is the next input character from path u.

GLOBAL sys1 -- sys2 S,I
GLOBAL marks the end of the local word name section and the start of word names. Must appear between **LOCAL** and **MODULE**. See also: **LOCAL MODULE**

GO -- S,E
Resumes execution of a word that was suspended by **BREAK** . See: **BREAK**

H' -- addr S "h-tick"
Used in the form: H' nnn
addr is the address of the first byte of the dictionary header for the next word in the input stream (nnn).

H>CODE addr1 -- addr2 S "h-to-code"
addr1 is the header address of a word in the dictionary, and addr2 is its corresponding code address. NOTE: addr2 is an absolute address.

HASH addr1 -- addr1 16b S

addr1 is the counted string address of a word and 16b is its corresponding hash code. *In FORTH09 this hash code is part of the word's header and is used to speed up dictionary searches.*

HERE -- addr 83R,S

The address of the next available dictionary location. *NOTE: In FORTH09 HERE returns a different value for each of the PRIMARY and SECONDARY code areas. (This is not specified in the standard).*

HEX -- 83C

Set the numeric input-output conversion base to sixteen.

HIDE -- S,I

Used in the form: HIDE nnn

Causes the header for the word nnn to be removed from the dictionary. The corresponding code for nnn is unaffected (not removed), but the word can no longer be found in the dictionary. The dictionary is packed, reducing its size by the size of the header for nnn. IMPORTANT: **HIDE** can not be used if a definition has been compiled to any vocabulary other than the vocabulary that nnn resides in, after the definition of nnn. Doing so will mess up the dictionary links.

HOLD char -- 83R

char is inserted into a pictured numeric output string. Typically used between <# and #> .

I -- w 83R,C

w is a copy of the loop index. May only be used in the form:

DO ... I ... LOOP or

DO ... I ... +LOOP

I$CHDIR addr1 u -- addr2

(pathlist_ptr access_mode -- new_pathlist_ptr)

addr1 is the pathlist string address and u is the access mode of the new directory for an I$CHDIR OS-9 system call. addr2 is the updated pathlist pointer.

I$CLOSE u --

(path# --)

u is the path number of an open path that will be closed.

I$CREATE addr1 u1 u2 -- addr2 u3

(pathlist_ptr permission access_mode --
pathlist_ptr+ path#)

addr1 is the pathlist name string address, u1 the file permissions (attributes), and u2 the access mode of a path to be created. addr2 is the updated pathlist pointer and u3 is the path number of the newly created path. NOTE: As in all OS-9 system calls **ERROR#** will be non-zero if an error occurred.

I$DELETE addr u --
(pathlist_ptr access_mode --)
The RBF file whose name string resides at addr will be deleted. NOTE: This call uses the newer I$DELETX system call so the access_mode u is used to determine whether the execution or data directory will be searched in the absence of an explicit pathname (one starting with /).

I$DUP u1 -- u2
(path# -- new_path#)
u1 is the path number to duplicate, and u2 is the duplicate path number.

I$MAKDIR addr u --
(pathlist_ptr attributes --)
addr is the address of the pathlist string, and u are the new directory attributes of the directory that will be created by the OS-9 I$MAKDIR call.

I$OPEN addr1 u1 -- addr2 u2
(pathlist_ptr access_mode -- pathlist_ptr+ path#)
addr1 is the address of the pathlist name string, and u1 is the access mode of the path to be opened. addr2 is the updated pathlist pointer and u2 is the path#.

I$READ u1 addr u2 -- u3
(max_cnt buffer_addr path# -- actual_cnt)
u1 is the maximum number of bytes to read from path u2 into buffer at addr with the OS-9 I$READ call. u3 is the actual number of bytes read.

I$READLN u1 addr u2 -- u3
(max_cnt buffer_addr path# -- actual_cnt)
u1 is the maximum number of bytes to read from path u2 into buffer at addr with the OS-9 I$READLN call. u3 is the actual number of bytes read.

I$SEEK d u --
(byte_position path# --)
d is the 32bit byte position and u is the path number for the I$SEEK call. The file will be positioned at d unless an error occurs.

I$WRITE u1 addr u2 -- u3
(max_cnt buffer_addr path# -- actual_cnt)
u1 is the maximum number of bytes to write to path u2 from buffer at addr with the OS-9 I$WRITE call. u3 is the actual number of bytes written.

I$WRITLN u1 addr u2 -- u3
(max_cnt buffer_addr path# -- actual_cnt)
u1 is the maximum number of bytes to write to path u2 from buffer at addr with the OS-9 I$WRITLN call. u3 is the actual number of bytes written.

IF flag -- 83R,S,C,I
-- sys (compiling)
Used in the form:
flag IF ... ELSE ... THEN
or
flag IF ... THEN
If flag is true, the words following **IF** are executed and the words following **ELSE** until just after **THEN** are skipped. The **ELSE** part is optional. If flag is false, words from **IF** through **ELSE** , or from **IF** through **THEN** (when no **ELSE** is used), are skipped. sys is balanced with its corresponding **ELSE** or **THEN** . *NOTE: FORTH09 provides the synonym ENDIF which may be substituted for THEN, making things more readable albeit non-standard.*

IFEND -- 83U,S,E "if-end"
Terminate a conditional interpretation sequence begun by **IFTRUE** . INTERPRET ONLY

IFF flag -- S,C,I "if-false"
-- sys (compiling)
Used in the form:
flag IFF ... ELSE ... ENDIF
or
flag IFF ... THEN
Performs the same function as **IF** except the polarity of the flag test is inverted. I.e. If flag is **FALSE** the code between **IFF** and **ENDIF** or **ELSE** will be executed. **IFF** replaces the sequence; 0= IF with shorter and faster code.

IFTRUE flag -- 83U,S,E "if-true"
INTERPRET ONLY.
Begin an
IFTRUE ... OTHERWISE ... IFEND
conditional sequence.
These words operate like **IF ELSE THEN** except that they cannot be nested, and are to be used only during interpretation. In conjunction with the words [and] they may be used within a colon definition to control compilation, although they are not to be compiled.

IMMEDIATE -- 83R,S
Marks the most recently created dictionary entry as a word which will be executed when encountered during compilation rather than compiled.

INDEX u1 u2 -- 83U,S
Print the first line of each screen over the range {u1..u2}. This displays the first line of each screen of source text which conventionally contains a title.

INLINE -- S
Used immediately after **END-CODE** to specify that the code of the preceding code word is to always be compiled inline. I.e. The machine code of the word is copied into the new word when it is compiled into a new word. The code definition should NOT end with an RTS instruction. See: **INLINEOK**

INLINEOK -- S

Used immediately after **END-CODE** to specify that the code of the preceding **CODE** word MAY be compiled inline, if the number of bytes of code is less than or equal to the value of **COMPLIMIT**, when the code word is later compiled into a new definition. The code definition MUST end with an RTS instruction, like all code definitions with the exception of **INLINE** word (See: **INLINE**).

INTERPRET -- 83C,S

Begin text interpretation at the character indexed by the contents of **>IN** relative to the block number contained in **BLK** , continuing until the input stream is exhausted. If **BLK** contains zero, interpret characters from the text input buffer.

J -- w 83R,C

w is a copy of the index of the next outer loop. May only be used within a nested DO-LOOP or DO-+LOOP in the form, for example:

DO ... DO ... J ... LOOP ... +LOOP

KEY -- 16b 83R

The least-significant 8 bits of 16b is the next ASCII character received. All valid ASCII characters can be received. Control characters are not processed by the system for any editing purpose. Characters received by **KEY** will not be displayed. NOTE: Only the least-significant 7-bits of a character may be used by a "standard program".

KEY? -- flag "key-question"

flag is true if a keyboard character (path 0) is waiting to be read.

L n -- S

One of the two really primitive line editor words. L displays line n of the last screen that was **LIST**ed, or otherwise has its screen number residing in the variable **SCR** (which shares storage with the VAR **SCRN**). Range of n is {0..15} See: **R**

LABEL: -- S,C,I "label-colon"

Used in the form: LABEL: nnn

Creates a dictionary entry for the word nnn. Used within a colon definition to create an entry point that can be referenced later as a word. The entry point cannot be placed where the return stack will be imbalanced as inside a DO-LOOP or between **>R** and **R>**. When **LABEL:** is encountered the dictionary entry for the original word being defined is closed so that it will be found in dictionary searches after the **LABEL:** . If the original word is to be **Immediate** or **componly** then the sequence: [IMMEDIATE] will have to be placed just before **LABEL:** in the definition.

LCOPY u1 u2 u3 u4 u5 -- S "L-copy"

(from_blk beg_line end_line dest_blk beg_line --)

LCOPY copies a range of lines from one screen to another. u1 is the source screen number, u2 is the first and u3 is the last of the range of lines to copy to the source screen to the destination screen u4 beginning at line u5 within the destination screen. E.g. to copy lines 2 thru 5 from screen 40 to lines 8 thru 11 of screen 50, 40 2 5 50 8 LCOPY . **LCOPY** does not affect the shadow screens. To copy the same range in the shadow screens you would also need to do: 40 SHADOW 2 5 50 SHADOW 8 LCOPY .

LEAVE -- 83R,S,C,I

-- (compiling)

Transfers execution to just beyond the next LOOP or +LOOP . The loop is terminated and loop control parameters are discarded. May only be used in the form:

DO ... LEAVE ... LOOP

or

DO ... LEAVE ... +LOOP

LEAVE may appear within other control structures which are nested within the do-loop structure. More than one **LEAVE** may appear within a do-loop. *NOTE: This definition of LEAVE differs from that in earlier FORTH dialects in that when LEAVE is encountered, a jump is made immediately out of the loop, versus older usage of changing the loop index so that the loop terminates the next time LOOP or +LOOP is encountered.*

LIST u -- 83C,S

The contents of screen u are displayed. The variable **SCR** (*and VAR **SCRN***) are set to u.

LITERAL -- 16b 83R,S,C,I

16b -- (compiling)

Typically used in the form:

[16b] LITERAL

Compiles a system dependent operation so that when later executed, 16b will be left on the stack.

LOAD u -- 83R,S

The contents of **>IN** and **BLK** , which locate the current input stream, are saved. The input stream is then redirected to the beginning of screen u by setting **>IN** to zero and **BLK** to u. The screen is then interpreted. If interpretation from screen u is not terminated explicitly it will be terminated when the input stream is exhausted and then the contents of **>IN** and **BLK** will be restored. An error condition exists if u is zero. See: **>IN BLK BLOCK**

LOCAL -- sys S,I,E

Used in the form:

LOCAL definitions GLOBAL definitions MODULE

marks the beginning of a local word name section of definitions. The header code for all words defined between **LOCAL** and **GLOBAL** will be removed from the dictionary when MODULE is executed. All definitions between LOCAL and MODULE must reside in the same vocabulary. See: GLOBAL MODULE

LOOP -- 83R,S,C,I

sys -- (compiling)

Increments the DO-LOOP index by one. If the new index was incremented across the boundary between limit-1 and limit the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues just after the corresponding **DO** . sys is balanced with its corresponding **DO** . See: **DO**

MAX n1 n2 -- n3 83R "max"
n3 is the greater of n1 and n2 according to the operation of > .

MAYBE 8b -- S,C,I
-- sys (compiling)
Used in the form:
c MAYBE c1 word1..cn wordn MAYBENOT
or
c MAYBE c1 word1 ...cn wordn
THENAGAIN ... MAYBENOT
MAYBE constructs a very fast and code efficient form of case statement. The 8b character c (least significant 8 bits of 16 bit word) are the case match criteria. A table of case characters c and word execution vectors is built. If c matches one of the case characters c1 .. cn then the associated word will be executed after which execution will continue after **MAYBENOT** . If there is no match then execution will continue after **MAYBENOT** unless the optional **THENAGAIN** clause is present in which case the words between **THENAGAIN** and **MAYBENOT** will be executed. **MAYBE** itself compiles the code between **MAYBE** and **THENAGAIN** or **MAYBENOT**, so there are restrictions on what may be used between these words. You may not comment in this area with ( ) or \ . The words between **MAYBE** and **THENAGAIN** must be matched pairs in the form:

n word where in is a numeric literal and word is a word name in the dictionary, or

ASCII c word where word is a word name in the dictionary.

MAYBENOT sys -- (compiling) S,C,I
marks the end of a **MAYBE** .. **MAYBENOT** construct, see **MAYBE** .

MEMLIMIT -- addr
addr is the address of the last byte of available data memory

MIN n1 n2 -- n3 83R "min"
n3 is the lesser of n1 and n2 according to the operation of < .

MOD n1 n2 -- n3 83R
n3 is the remainder after dividing n1 by the divisor n2. n3 has the same sign as n2 or is zero. An error condition results if the divisor is zero or if the quotient falls outside of the range {-32,768..32,767}.

MODULE sys -- (compiling) S,I,E
Used in the form:
LOCAL worddefs GLOBAL worddefs MODULE
Execution of **MODULE** causes the dictionary entries for all words defined between **LOCAL** and **GLOBAL** to be removed from the dictionary. Only the dictionary headers are removed, the actual code for all entries is unaffected. All words defined between **LOCAL** and **MODULE** must reside in the same vocabulary otherwise an error condition exists. See: **LOCAL GLOBAL**

MOREBLOCK u -- S
The current block storage path and its associated shadow block path are extended by u screens of spaces.

NEGATE n1 -- n2 83R
n2 is the two's complement of n1, i.e., the difference of zero less n1.

NIP 16b1 16b2 -- 16b2
16b1 is remove from the stack. Equivalent to SWAP DROP , but shorter and faster.

NOT 16b1 -- 16b2 83R
16b2 is the one's complement of 16b1.

NUMBER? addr -- d tf
or addr -- ff
addr is the address of a counted string. If the string can be converted to a number in the current number **BASE** then d will be the converted value and tf will be a flag with value=TRUE, otherwise ff will be a flag with value=FALSE.

ONLY -- 83E,S
Select just the **ONLY** vocabulary as both the transient vocabulary and resident vocabulary in the search order. *NOTE: In FORTH09 the ONLY vocabulary is a subset of the FORTH vocabulary, i.e. the ONLY words will be in the search order any time FORTH is in the search order even if SEAL is used to remove ONLY.*

OP0 -- S "op-zero"
Used in defining words, **OP0** sets a flag to instruct the compiler that the last compiled code was neither a bsr or lbsr instruction. The compiler uses this information when optimizing and if this is not done by a word that has compiling action (ending with , or **C,**) then invalid code may be produced, which may crash.

OR 16b1 16b2 -- 16b3 83R
16b3 is the bit-by-bit inclusive-or of 16b1 with 16b2.

ORDER -- 83E,S
Displays the vocabulary names forming the search order in their present search order sequence. Then shows the vocabulary into which new definitions will be placed.

OTHERWISE -- 83U,S,E
An interpreter-level conditional word. See: **IFTRUE**

OVER 16b1 16b2 -- 16b1 16b2 16b3 83R
16b3 is a copy of 16b1.

PAD -- addr 83R
The lower address of a scratch area used to hold data for intermediate processing. *The address or contents of PAD may change and the data may be lost if the address of the next available dictionary location is changed.* The minimum capacity of **PAD** is 84 characters. *NOTE: The above italicized statement does NOT apply to FORTH09.*

PICK +n -- 16b 83R
16b is a copy of the +nth stack value, not counting +n itself. {0 ..the number of elements on stack-1}. 0 PICK is equivalent to DUP,
1 PICK is equivalent to OVER .

PRIMARY -- S
Changes the value of the **VAR PSTATE** so that subsequent definitions will be compiled to the **PRIMARY** (application) area of the memory map. These can later be saved as independent executable modules with **SAVE** . See: **SECONDARY**

PSTATE -- n S

A **VAR** whose value of zero indicates the **PRIMARY** state, and system dependent non-zero value indicates **SECONDARY** state. See discussion of special FORTH09 features elsewhere in this document.

PUTCH u1 u2 --

(char path# --)

The least significant 8 bits of u1 is written to path number u2.

QUIT -- 83R,S

Clears the return stack, sets interpret state, accepts new input from the current input device and begins text interpretation. No message is displayed.

R +n -- S

Used in the form:

n R text

A really primitive line editor word which replaces line +n of that most recently LISTed screen with text. The remainder of the input line following the first space after **R** is considered the text, and is padded to 64 characters with spaces.

R0 -- 16b S "s-zero"

16b is the initial value of the return stack pointer. (I.e. the value when the return stack is empty)

R> -- 16b 83R,C "r-from"

16b is removed from the return stack and transferred to the data stack.

R@ -- 16b 83R,C "r-fetch"

16b is a copy of the top of the return stack.

RDROP -- C "r-drop"

(return stack) 16b --

16b is dropped from the return stack, equivalent to: R> DROP but much shorter.

READ addr u1 u2 --

(buffer_addr count path# --)

u1 bytes are read from path number u2 and transferred to buffer at addr. NOTE: The OS-9 I$READ call is used for this read. The actual number of bytes read is stored in the variable SPAN.

READLN addr u1 u2 -- "read-line"

(buffer_addr count path# --)

u1 bytes are read from path number u2 and transferred to buffer at addr. The OS-9 I$READLN call is used for this read, so less than count bytes may be read if a carriage return is encountered. The actual number of bytes read is stored in the variable **SPAN**.

READLN0 addr u -- "read-line-zero"

(buffer_addr count --)

u bytes are read from path 0 using the OS-9 I$READLN call and stored into the buffer at addr. Less than u bytes may be read if a carriage return is encountered. The actual number of bytes read will be stored into the variable **SPAN** . This is equivalent to **EXPECT** with the exception that a carriage return is echoed as a carriage return instead of a space.

READY? u -- flag "ready-question"
(path# -- flag)
flag will be true if the open SCF path number u has a character waiting to be read.

RECURSE -- 83C,S,C,I
-- (compiling)
Compile the compilation address of the definition being compiled to cause the definition to later be executed recursively.

REMEMBER -- 83U,S
A defining word executed in the form:
REMEMBER <name>
Defines a word which when executed, will cause <name> and all subsequently defined words to be deleted from the dictionary. <name> may be compiled into and executed from a colon definition. The sequence DISCARD REMEMBER DISCARD provides a standardized preface to any group of transient word definitions.

REPEAT -- 83R,S,C,I
sys -- (compiling)
Used in the form:
BEGIN ... flag WHILE ... REPEAT
At execution time, **REPEAT** continues execution to just after the corresponding **BEGIN** . sys is balanced with its corresponding **WHILE** . See: **BEGIN**

REV n -- S
The module revision number of the next module saved by **SAVE** , **XSAVE** , or **SAVESYS** will be set to n.

RMB -- addr S "r-m-b"
A defining word used in the form:
u RMB <name>
creates a new word definition <name> that when later executed will leave addr on the stack which will be the address of u bytes of variable data storage. This is equivalent to the RMB 6809 assembler directive. The sequence 2 RMB is equivalent to **VARIABLE**, and 4 RMB is equivalent to **2VARIABLE**. The addr of storage left on the stack will be the absolute data storage address. This is calculated at runtime from an offset into the data area.

ROLL +n -- 83R
The +nth stack value not counting +n itself is first removed and then transferred to the top of the stack, moving the remaining values into the vacated position. {0..the number of elements on the stack-1}
2 ROLL is equivalent to **ROT**
0 ROLL is a null operation

ROT 16b1 16b2 16b3 -- 16b2 16b3 16b1 83R "rote"
The top three stack entries are rotated, bringing the deepest to the top.

RP! 16b -- S "r-p-store"
16b becomes the new return stack pointer value. **Use with extreme caution!**

RP@ -- 16b S "r-p-fetch"
16b is the return stack pointer value.

S. n1 n2 ... nx -- S,I "s-dot"
All elements on the stack are removed and displayed in the current number base. The deepest element is displayed first. All items on the stack are assumed to be 16 bits. If the stack is empty then the message "empty" is displayed. See: **.S**

S0 -- addr "s-zero"
addr is the address of the bottom of the stack.

S>D n -- d "s-to-d"
d is the result of sign extending n.

SAVE -- <name> S
The **PRIMARY** application code up through the word <name> will be saved as an OS-9 executable module with the pathname also equal to <name>. The word <name> will be the execution entry point. Any user variables will be uninitialized when the program is run. **BASE** will have the value it contained at the time the **SAVE** was done. See: **XSAVE EDITION REV**

SAVE-BUFFERS -- 83R,S "save-buffers"
The contents of all block buffers marked as **UPDATE**ed are written to their corresponding mass storage blocks. All buffers are marked as no longer being modified but may remain assigned.

SAVESYS -- <name> S
The FORTH09 development system including all **LOAD**ed definitions will be saved to the pathname <name> in the current directory. This complete system can later be loaded with the OS-9 command FLOAD <name> which saves the time that would otherwise be required to **LOAD** all the word definitions. **SAVESYS** cannot however be run after any new vocabulary has been defined, only the predefined vocabularies, **FORTH**, **ONLY**, and **ASSEMBLER** may be present.

SCOPY u1 -- u2 S "s-copy"
The **SHADOW** screen for block u1 is copied to the **SHADOW** screen for block u2.

SCR -- addr 83C,S "s-c-r"
The address of a variable containing the number of the screen most recently **LIST**ed. See: **SCRN**

SCRN -- u S "s-c-r-n"
SCRN is a **VAR** which holds the number of the last screen displayed by **LIST**. It shares storage with the variable **SCR** so writing to one affects the contents of the other.

SCRWIDTH -- +n S "screen-width"
SCRWIDTH is a **VAR** whose contents, +n describe the width of the display screen. This information is used by the word **WORDS** to determine how long of a line it may display. This is a **SECONDARY VAR** so its contents will be preserved by **SAVESYS**.

SEAL -- 83E,S
Delete all occurrences of **ONLY** from the search order. The effect is that only specified application vocabularies will be searched.

SECONDARY -- S

After executing **SECONDARY**, subsequent word definitions will be made to the secondary (non-application) code area and their headers chained to the **SECONDARY** thread of the current vocabulary.

SET.OPT addr u -- "set-options"

(buffer_addr path# --)

Sets the path options in the 32 byte buffer at addr to path number u. Uses the OS-9 I$SETSTT call of SS.OPT .

SHADOW u1 -- u2 S

u2 is the shadow screen number corresponding to normal screen u1. NOTE: Shadow screens occupy a separate file from the normal screens, u2 = u1 **OR**ed with 8000 (hex), which flags the various block words to access the shadow file.

SHELL addr u --

addr is the address and u is the size of the parameter field that is forked to the program shell. The parameter string at addr must be terminated with a carriage return character. See: **CR"**

SHIFT 16b1 n -- 16b2 83U

Logical shift 16b1 left n bits if n is positive, right n bits if n is negative. Zeros are shifted into vacated bit positions.

SHOW u1 u2 -- S

The screens beginning with u1 and ending with u2 in the current block storage file are displayed to the printer. The VAR 132COL? determines if shadow screens will be displayed along side of normal screens. If 132COL? is FALSE only the normal screens will be displayed. The pathname for the printer may need to be edited by the user for system dependencies.

SHOWS u1 u2 -- S "show-s"

The SHADOW screens only beginning with u1 and ending with u2 in the current block storage file will be listed to the printer. See: SHOW

SIGN n -- 83R

If n is negative, an ASCII "-" (minus sign) is appended to the pictured numeric output string. Typically used between **<#** and **#>** .

SIGN_ -- "sign-underscore"

If the number whose conversion was started by <# is negative then an ASCII "-" is appended to the pictured numeric output string. Similar to **SIGN** except the sign bits of the number being converted need not be on the stack, since they are stored in an internal variable.

SLIDE u1 u2 n -- S

The range of mass storage blocks u1 thru u2 is exchanged with the range of blocks u1+n thru u2+n. No blocks are erased by this process. The associated shadow blocks are also exchanged.

SP@ -- addr 83C "s-p-fetch"

addr is the address of the top of the stack just before **SP@** was executed.

SPACE -- 83R

Displays an ASCII space.

SPACES +n -- 83R

Displays +n ASCII spaces. Nothing is displayed if +n is zero.

SPAN -- addr 83R

The address of a variable containing the count of characters actually received and stored by the last execution of **EXPECT** . See: **EXPECT**

STATE -- addr 83R,S

The address of a variable containing the compilation state. A non-zero content indicates compilation is occurring, but the value itself is system dependent. A Standard Program may not modify this variable.

SWAP 16b1 16b2 -- 16b2 16b1 83R

The top two stack entries are exchanged.

SWAP! addr 16b -- "swap-store"

16b is stored at addr. This is equivalent in function to the separate operations: SWAP ! but because of 6809 architectural reasons is faster than doing either SWAP or a ! operation by itself.

THEN -- 83R,S,C,I

sys -- (compiling)

Used in the form:

flag IF ... ELSE ... THEN

or

flag IF ... THEN

THEN is the point where execution continues after **ELSE**, or **IF** when no **ELSE** is present. sys is balanced with its corresponding **IF** or **ELSE** . See: **IF ELSE** (and non-standard synonym **ENDIF**).

THENAGAIN sys -- sys (compiling) S,C,I

Used in the form:

MAYBE ... THENAGAIN ... MAYBENOT

marks the end of the conditional case table and the beginning of the alternate case words. Words between **THENAGAIN** and **MAYBENOT** will be executed only if none of the cases between MAYBE and THENAGAIN matched. See: MAYBE MAYBENOT

THRICE -- S,C,I

Used in the form:

: <name> ... THRICE ... ;

Causes the words between **THRICE** and **;** to be executed a total of 3 times. This is a controlled form of recursion. See: **TWICE**

THRU u1 u2 -- 83C,S

Load consecutively the blocks from u1 through u2.

TIB -- addr 83R,S "t-i-b"

The address of the text input buffer. This buffer is used to hold characters when the input stream is coming from the current input device. The minimum capacity of **TIB** is 80 characters.

TO --

Used in the form:

TO <varname>

Cause the **VAR** <varname> to have the action of removing a 16 bit number from the stack and storing it in its variable storage rather than fetching it.

TOFLAG -- n S "to-flag"

n is the value of the **VAR TOFLAG** which determines whether a **VAR** will fetch or store its value when referenced.

TRUE -- -1

A constant with a value of -1, or true when used as a flag.

TUCK 16b1 16b2 -- 16b2 16b1 16b2

Equivalent in operation to SWAP OVER but faster than **SWAP** alone.

TWICE -- S,C,I

Used in the form:

: <name> ... TWICE ... ;

causes the words between **TWICE** and **;** to be executed twice. More efficient that creating a separate word definition and referencing it twice. See: **THRICE**

TYPE addr +n -- 83R

+n characters are displayed from memory beginning with the character at addr and continuing through consecutive addresses. Nothing is displayed if +n is zero.

U. u -- 83R "u-dot"

u is displayed as an unsigned number in a free-field format.

U.R u +n -- 83C "u-dot-r"

u is converted using the value of **BASE** and then displayed as an unsigned number right aligned in a field +n characters wide. If the number of characters required to display u is greater than +n an error condition exists.

U< u1 u2 -- flag 83R "u-less-than"

flag is true is u1 is less than u2.

U>= u1 u2 -- flag "u-greater-or-equal"

flag is true if u1 is greater than or equal to u2.

UM* u1 u2 -- ud 83R "u-m-times"

ud is the unsigned product of u1 times u2. All values and arithmetic are unsigned.

UM/MOD ud u1 -- u2 u3 83R "u-m-divide-mod"

u2 is the remainder and u3 is the floor of the quotient after dividing ud by the divisor u1. All values and arithmetic are unsigned. An error condition results if the divisor is zero or if the quotient lies outside the range {0..65,535}.

UNTIL flag -- 83R,S,C,I
sys -- (compiling)
Used in the form:
BEGIN ... flag UNTIL
Marks the end of a BEGIN-UNTIL loop which will terminate based on flag. If flag is true the loop is terminated. If flag is false execution continues to just after the corresponding **BEGIN** . sys is balanced with its corresponding **BEGIN** . See: **BEGIN**

UPDATE -- 83R,S
The currently valid block buffer is marked as modified. Blocks marked as modified will subsequently be automatically transferred to mass storage should its memory buffer be needed for storage of a different block or upon execution of **FLUSH** or **SAVE-BUFFERS**.

USING -- S
Used in the form:
USING <pathname>
Closes the current mass storage path and attempts to open <pathname> as the new path. Also attempts to open pathname.s as the shadow path.

VAR -- S
Used in the form:
VAR <name>
creates a new dictionary entry for <name>. Later execution of name will either fetch a 16 bit variable to the stack or remove a 16 bit variable from the stack and store it in its allocated data space depending on the contents of the system var **TOFLAG**. If **TOFLAG** is zero then executing <name> will fetch the variable contents. If **TOFLAG** is non-zero then the variable contents will be stored. **TOFLAG** is set non-zero by preceding the <name> with the word TO . **TOFLAG** is automatically cleared when the var value is stored.

VARIABLE -- 83R,S
A defining word executed in the form:
VARIABLE <name>
A dictionary entry for <name> is created and two bytes are **ALLOT**ed in its parameter field. This parameter field is to be used for contents of the variable. The application is responsible for initializing the contents of the variable which it creates. When <name> is later executed, the address of its parameter field is placed on the stack. *NOTE: In FORTH09 the variable parameter field is in the process data memory, not in the code memory.*

VOCABULARY -- 83R,S
A defining word executed in the form:
VOCABULARY <name>
A dictionary entry for <name> is created which specifies a new ordered list of word definitions. Subsequent execution of <name> replaces the first vocabulary in the search order with <name>. When <name> becomes the compilation vocabulary new definitions will be appended to <name>'s list.

WHERE -- S
After an error abort is taken during a **LOAD** operation, **WHERE** will display the block number where the error occurred (in the current number base) and display the portion of the block that was processed up to the point where the error was detected.

WHILE flag -- 83R,S,C,I
sys1 -- sys2 (compiling)
Used in the form:
BEGIN ... flag WHILE ... REPEAT
Selects conditional execution based on flag. When flag is true, execution continues to just after the **WHILE** through to the **REPEAT** which then continues execution back to just after the **BEGIN** . When flag is false, execution continues to just after the **REPEAT** , exiting the control structure. sys1 is balanced with its corresponding **BEGIN** . sys2 is balanced with its corresponding **REPEAT** . See: **BEGIN**

WIDTH -- n S
or TO n --
WIDTH is a **VAR** which contains the count of the maximum number of word name characters to store in the dictionary for subsequent word definitions. E.g. if **WIDTH** is set to 3 then only the first 3 characters of word names will be stored in future definitions. Duplicate word names will be extremely rare even for a **WIDTH** of 1 however because of the hash code that is used during searching.

WLIST u -- S
Displays screen u side by side with its shadow screen in 132 column screen mode. Strike any key after the display is finished to return to 80 column screen mode. NOTE: This word is only defined if you load a version of **EDIT** for a terminal supporting 132 column displays.

WORD char -- addr 83R,S
Generates a counted string by non-destructively accepting characters from the input stream until the delimiting character char is encountered or the input stream is exhausted. Leading delimiters are ignored. The entire character string is stored in memory beginning at addr as a sequence of bytes. The string is followed by a blank which is not included in the count. The first byte of the string is the number of characters {0..255}. If the string is longer than 255 characters, the count is unspecified. If the input stream is already exhausted as **WORD** is called, then a zero length character string will result.

If the delimiter is not found the value of **>IN** is the size of the input stream. If the delimiter is found **>IN** is adjusted to indicate the offset to the character following the delimiter. **#TIB** is unmodified.

The counted string returned by **WORD** resides in the "free" dictionary area at **HERE** . Note that the text interpreter may also use this area.

WORDFIND -- addr flag S
or -- addr u flag
Used in the form:
WORDFIND <name>
Reads the next word (delimited by a space) from the input stream and searches the dictionary for it. If the word is not found then flag is false, addr is the counted string address of the word name, and u is its hash code. If the word is found then flag is true and addr is the header address of the word. (See: **H'**)

WORDS -- 83E,S
List the word names in the first vocabulary of the currently active search order.

WRITE addr u1 u2 --
(buffer_addr count path# --)
u1 bytes are written to path u2 from the buffer at memory address addr using the OS-9 I$WRITE system call. After the call the var **ERROR#** will be non-zero if any error occurred.

WRITELN addr u1 u2 -- "write-line"
(buffer_addr count path# --)
u1 bytes are written to path u2 from the buffer at memory address addr using the OS-9 I$WRITLN system call. After the call the var **ERROR#** will be non-zero if any error occurred.

XOR 16b1 16b2 -- 16b3 83R "x-or"
16b3 is the bit-by-bit exclusive-or of 16b1 with 16b2.

XSAVE u -- S "x-save"
Used in the form:
u XSAVE <name1> <name2>
Performs the function of **SAVE** with the added features that extra data memory can be specified for the saved module and the module name can differ from the word name saved. name1 is the word name to be saved, and name2 is the pathname where it will be saved (defaults to execution directory). u bytes will be added to the current data memory allocation and the total used as the minimum data memory for the saved module. See: **SAVE**

[-- addr 83R,S,I "left-bracket"
-- (compiling)
Sets interpret state. The text from the input stream is subsequently interpreted. See:]

['] -- addr 83R,S,C,I "bracket-tick"
-- (compiling)
Used in the form:
['] <name>
Compiles the compilation address addr of <name> as a literal. When the colon definition is later executed addr is left on the stack. An error condition exists if <name> is not found in the currently active search order.

[A -- S,I "left-bracket-a"
Switches state to execution and executes **ASSEMBLER** making it the first vocabulary in the search order. Using **[A** within a definition effectively switches from hi level word compilation to inline code compilation. The associated word **F]** switches back.

[COMPILE] -- 83R,C,I "bracket-compile"
-- (compiling)
Used in the form:
[COMPILE] <name>
Forces compilation of the following word <name>. This allows compilation of an immediate word when it would otherwise have been executed.

**** -- S,I "backslash"
Used in a mass storage screen, \ makes the remainder of the line a comment by incrementing the value of **>IN** to the start of the next line.

] -- 83R,S "right-bracket"
Sets compilation state. The text from the input stream is subsequently compiled. See: [

const_ -- 16b
The runtime routine for **CONSTANT** . Moves the 16b word stored immediately after the compiled const_ by , (comma) to the stack.

doff -- 16b S "data-offset"
A **VAR** that contains the offset from the start of the data area to the next free byte of data memory. The value of **doff** is altered by **ALLOT_D** . **doff** is a temporary working variable whose value will go into effect after executing ; or **(;)** . It will be reset by ABORT when a definition fails.

edition -- 16b S
16b is the edition number of the FORTH09 system. This is not the same as the release number which represents the number of the current commercial release. edition gets changed with each edit of the Forth module code, so do not be alarmed if it is a rather high number.

maybe_ 8b --
This is the runtime routine for **MAYBE**.

pfpush -- addr "parameter-field-push"
This is the runtime that gets called by **DOES>** . It moves the return address from the call to pfpush from the return stack to the data stack.

prompt$ -- addr "prompt-string"
prompt$ is a variable that contains the address of a counted string that is used as the interpreter prompt for keyboard input. The default prompt is ">" . addr is the variable storage address. The value at this address is the address of a byte containing the character count of the string, with the first character of the string starting at address+1.

rmb_ -- addr
The runtime used by **RMB** . Used in the form:
COMPILE rmb_ ,
the word compiled by , immediately after **rmb_** is a data offset into the data memory area. rmb_'s runtime action is to compute the absolute memory address of the beginning of the data area plus the offset and push this on the stack.

strlit_ -- addr u
String literal runtime routine. A counted string is compiled immediately after **strlit_** . At runtime **strlit_** reads the count byte of the string and pushes it's address addr and count u on the stack.

var_ -- 16b or
16b -- (if **TOFLAG** set)
The runtime for **VAR**s. The variable data offset is compiled immediately after the compilation of **var_** .

FORTH LANGUAGE OVERVIEW

This section will introduce you to the operation of the Forth language in general. This is not meant to be a complete instructional text on Forth, but rather a brief outline of how Forth operates. This may be enough for an experienced programmer to get a start with Forth, but a novice will almost surely require an additional beginning text as suggested in chapter one. References to features specific to FORTH09 and not commonly found in other Forth-83 languages will be *italicized*. Sequences of Forth code will be underlined. Forth words will be **highlighted.** Since the best way to learn a new language is with the system running in front of you, so you can immediately try examples, you should install FORTH09 on your system as described in chapter 1 before proceeding too far here.

WORDS

The fundamental programming unit in Forth is the "word". A Word can be thought of as a subroutine, but it also encompasses what are normally considered operators in other languages. Most other languages have a set of operators (such as + - * etc.) and programming keywords, which are finite in number. Outside of these you program by using groups of these operators and keywords in subroutines or functions. In Forth you actually extend the language as you program.

A "word name" in Forth is composed of a one or more ASCII characters delimited by spaces. There are no restrictions on what characters can be used.

Examples of words: **SWAP DROP + @ 1+**

The glossary section of this manual defines the action of each Forth word provided with the system. You program by defining new words in terms of those that already exist. You can in turn define new words in terms of the ones you have defined, building up your code pyramid fashion until a complex function is defined as a single word.

STACKS

Most programmers are familiar with the concept of a "stack". A stack is an area in memory, that has a pointer called a "stack pointer" associated with it. The stack pointer, points to the next available data location on the stack. You place data on the stack with a special "push" operation that automatically places your data item in the next available location and then advances the stack pointer to point to the next location. To remove the data there is a corresponding "pull" (sometimes called "pop") operation which first "backs up" the stack pointer to the last used location and removes the data at that location. From this it can be seen that if you "push" several items onto the stack and then remove them, the last item pushed will be the first item pulled. (i.e. we have a LIFO "Last In First Out" stack). The advantage of this over a simple array to hold data is that you never need to know exactly where in memory (e.g. what array subscript) the data is. The stack pointer takes care of locating the data automatically.

Forth uses TWO stacks. The customary "return stack" is used as in other systems to hold the return address for subroutine calls. The data stack holds data or parameters to be operated on by Forth words. When we refer to "the stack" in Forth we are talking about the **Data Stack** unless we specifically refer to the **Return Stack**.

All Forth words that operate on data take their operands from the stack (data stack) and return their results on the stack. For example the word **+** ("plus") performs the addition function, adding two numbers together. When **+** ("plus") executes it removes two numbers from the stack, adds them together, and places the resulting sum back on the stack. The word **.** ("dot" the ASCII period) removes one number from the stack and displays its value on the console screen.

> Enter FORTH09 as described on the first page of chapter one and try typing the following. (End each line by typing the "return" key).
>
> 3 . (Forth responds with **3**)
>
> (This places the number 3 on the stack and then displays it).

2 5 + . (Forth responds with **7**)

(Place the number 2 and then 5 on the stack, add them together, and display the result).

You can see from these examples that Forth uses what is known as "reverse polish" or "postfix" notation, similar to HP calculators. That is, rather that writing "2 + 5" you enter "2 5 +". The operands precede the operator.

The math words - * / operate in a similar fashion to + , i.e. each gets its two operands from the stack, performs the operation and places its single result on the stack. All of these operators deal with 16-bit integer numbers, which is the default word size in Forth. Try some experiments with these:

20 4 - 3 *

This is equivalent to (20 - 4) * 3 in more familiar mathematical notation. Note that the use of postfix notation eliminates the need for parenthesis in the expression. Print the result from the stack with . ("dot"). Forth should display **48**.

The FORTH09 word *.S* ("dot-s") displays the contents of the stack in the order in which the elements were placed there, without removing anything from the stack. Experiment with this word.

What happens if you attempt a "dot" operation and nothing is on the stack? Answer: You underflow the stack *and get a warning message stating such.* Every number you put on the stack should come off at some point. It is important to keep the stack "balanced" at all times. If you were in a loop that always put one number more on the stack than it took off, you would soon overflow the limits of the stack and in fact would probably overwrite all of memory and crash the system requiring a reboot.

INPUT STREAM

The "input stream" is a term in Forth that refers to the stream of characters being input to the system by you from the keyboard. The input stream may be redirected at times to mass storage (disk file) or other sources. As you have been typing into Forth, the Forth interpreter is picking a word at a time from the input stream and then deciding what to do with it. The Forth interpreter does not look at the line until after you type the "return" at the end. Forth then skips over any leading ASCII space characters and considers a "word" to be any string of non-space characters. After Forth has found a word in the input stream it then looks in its dictionary to see if the word is defined.

DICTIONARIES

Forth keeps a list of all previously defined "words" in a dictionary. The dictionary contains that word name and other information defining the action of the word. Every time Forth picks a word out of the input stream it searches the dictionary to see if the word is there. If it finds the word in the dictionary, Forth executes the word (i.e. performs the word's function by running the code associated with that word). If the word is not found in the dictionary, then Forth attempts to interpret the word as a number. If the word was the ASCII representation of a number, then the conversion will be successful, and Forth will place the value of the number on the stack. If the number conversion was not successful, then a message is displayed indicating that the word was not found in the dictionary.

So you can now see what has been happening when you type. For instance, if you typed 123 . Forth would first pick up the "word" **123** which it would not find in the dictionary. It would then try to convert **123** to a number, which succeeds, and then place it on the stack. Next Forth picks up the word . ("dot"), which is found in the dictionary, so the code for "dot" gets executed. The code for "dot" removes the number **123** from the stack and displays it on your terminal screen.

STATES

If what we have just described was all that Forth did, then it would be little more than a calculator. This is not quite the whole picture though. Forth has two "states". Forth is always in one of these states (or modes). What we described previously was exactly what happens in the "interpret" state. The other state is the "compile" state. There is a word named **STATE** which is a variable which flags

the system whether it should be interpreting or compiling. The "compile" state is used in creating new word definitions. When Forth picks a word out of the input stream and finds it in the dictionary, it then checks the value of **STATE** . If Forth is interpreting it executes the word as explained previously, but if Forth is compiling, it then "compiles" the word into the new definition. If the word was not in the dictionary, but converted as a number, and Forth is compiling, then a "literal" number is compiled into the definition. When the new word is later executed, the "literal" number is placed on the stack at that time.

COLON DEFINITIONS

The Forth word : ("colon") is used to create new word definitions. You program in Forth by defining new words with "colon", which are called "colon definitions".

Example: : HI ." HELLO OUT THERE" ;

Defines the new word **HI** . When you execute **HI** (by typing **HI** on the keyboard followed by a return-key), **HI** will display the message "HELLO OUT THERE" on the screen. This example uses two additional words that we haven't discussed yet, ." ("dot-quote"), and ; ("semi-colon").

When the word : ("colon") executes, it takes the next word from the input stream, which in our example is the word "HI" and begins a definition for that word. A dictionary entry for HI is made and Forth is switched to the compilation state. The words in the input stream are compiled into the definition for HI until the ; ("semi-colon") is encountered.

PRECEDENCE

Actually there is one more exception to Forth's action in the compile state. (This is the last one we swear!). Each word has a bit of information in its dictionary entry, called the "precedence bit". If this bit is set, the word is said to be an **IMMEDIATE** word. If Forth is in the compile state, and it finds a word in the dictionary that is an IMMEDIATE word, Forth **executes that word immediately instead of compiling it.**

In our last example, you may have wondered how the text message "HELLO OUT THERE" was compiled by Forth. The First word the compiler encountered in the definition was ." ("dot-quote"). Notice the space after "dot-quote" distinguishing it from the text message. Dot-quote is an immediate word, so instead of compiling, Forth executes it ("immediately"). The action of dot-quote is to read characters from the input stream up until a " ("closing quote") is found, and compile those characters as a string literal in the current definition, such that when that definition is executed those characters will be displayed.

The "semi-colon" is also an immediate word. Its action is to stop compiling, make the word just compiled accessible in the dictionary, and switch Forth back to the interpret state. The interpret state is also referred to as the "execute" or "executing" state).

If you tried to type in ." something" outside of a new colon definition, you will have noticed that *FORTH09 displays the message "--compilation only"*. This indicates that the word dot-quote is only allowed within a colon definitions, i.e. when Forth is in the compile state. There are a number of Forth words that are only allowed to be used in the compilation state. Most of the IMMEDIATE words are also COMPILATION ONLY words. If you look in the glossary section of the manual, you will notice a code on the first line of each word definition. An **I** indicates the word is IMMEDIATE, and an **C** indicates it is Compilation only. If you want Forth to echo back a message while you are in the interpret state, there is a word similar to dot-quote, namely "dot-paren".

Example: .(something)

Displays the message we tried earlier. Note the space after .(which is not a part of the actual message. The message characters begin after the first space following .(and end with the character just before the) ("closing-paren").

DEFINING WORDS

"Colon" is a defining word. A defining word is a word that defines other words. That is, it creates a dictionary entry for a new word that can later be found in the dictionary and executed or compiled. In Forth you can create new defining words that can later be used to create new words.

VARIABLES

The word **VARIABLE** , is a Forth defining word used to create integer variables.

```
VARIABLE MYNUMBER
```

Creates a new word "**MYNUMBER**". When you later execute the word **MYNUMBER**, it leaves an address on the stack which points to 2 bytes of variable storage that may be written or read.

```
MYNUMBER @
```

The word @ ("fetch"), takes an address from the stack, reads or "fetches" the value (2-bytes) at that address, and places that value on the stack. Note that Forth variables are uninitialized, i.e. their initial value is not specified.

To write a value to the variable, the word **!** ("store") is used. "Store" takes two numbers off the stack. The first is the value to be written, and the second is the address of where to write the first value.

```
347 MYNUMBER !
```

Sets the value of **MYNUMBER** to 347. We have assumed so far that numbers are input in decimal (base 10), but Forth allows the use of any number base between 2 and 72 for input and output of numbers. The **VARIABLE BASE** contains the value of the number base in current use. If the value of **BASE** is **10**, then input is in decimal, **16** indicates hexadecimal input, etc.

Great care should be exercised when using the word **!** and any other words that alter (write to) memory. If the top most element on the stack is not a valid address to store to, the operation will still take place using what ever value was on the stack as an address. Something will always be written somewhere!. It is quite possible (easy) to crash your system with improper use of this word. Any time you suspect you have done so, your system should be rebooted to avoid possible improper operations due to some code somewhere being clobbered.

CONTROL STRUCTURES

Forth contains a set of words to implement looping, and conditional execution. The word **IF** which is common to most computer languages, implements conditional execution of words. **IF** is an **IMMEDIATE** word that can only be used within a colon definition. The word **IF** removes a single data word from the stack which it interprets as a flag. If the flag value was zero, it is considered a FALSE value, and execution of the words between **IF** and **ENDIF** is skipped. If the flag value was non-zero, then it is considered TRUE, and execution continues with the words following **IF**. The optional word **ELSE** may appear between **IF** and **ENDIF** and implements its universally assumed function.

e.g.
```
: WHAT IF ." ITS TRUE" ELSE ." ITS FALSE" ENDIF ;
0 WHAT                    Forth responds ITS FALSE
1 WHAT                    Forth responds ITS TRUE
```

Actually the standard word for the **ENDIF** function is **THEN** , which you may use, but we prefer to use the word **ENDIF**, which has the same function, since it looks less confusing.

Looping is implemented with the words **DO** and **LOOP** or **+LOOP**, for a loop structure similar to the FOR-NEXT loop in basic. A loop with a test at the end is constructed with **BEGIN** and **UNTIL**. A loop with a test at the beginning can be constructed with **BEGIN WHILE** and **REPEAT**. Refer to the glossary or a text book for details on these.

MASS STORAGE

The only form of mass storage defined in the Forth-83 standard is the **BLOCK** storage. A block is a 1K (1024 byte) section of mass storage (disk sector[s]) that can contain Forth program code or data. Blocks are numbered zero through n (n being the number of blocks available minus one). The word **LOAD** redirects the input stream to a specific block. The contents of the block are then handled by Forth exactly as if they were typed in on the console as one big (1024 byte) line. Forth usually provides some form of editor to edit the contents of the storage blocks. In general you edit a number of blocks to contain new Forth words, and **LOAD** them in for testing and execution. They can be erased from memory with the word **FORGET** and reloaded as testing and program development proceed.

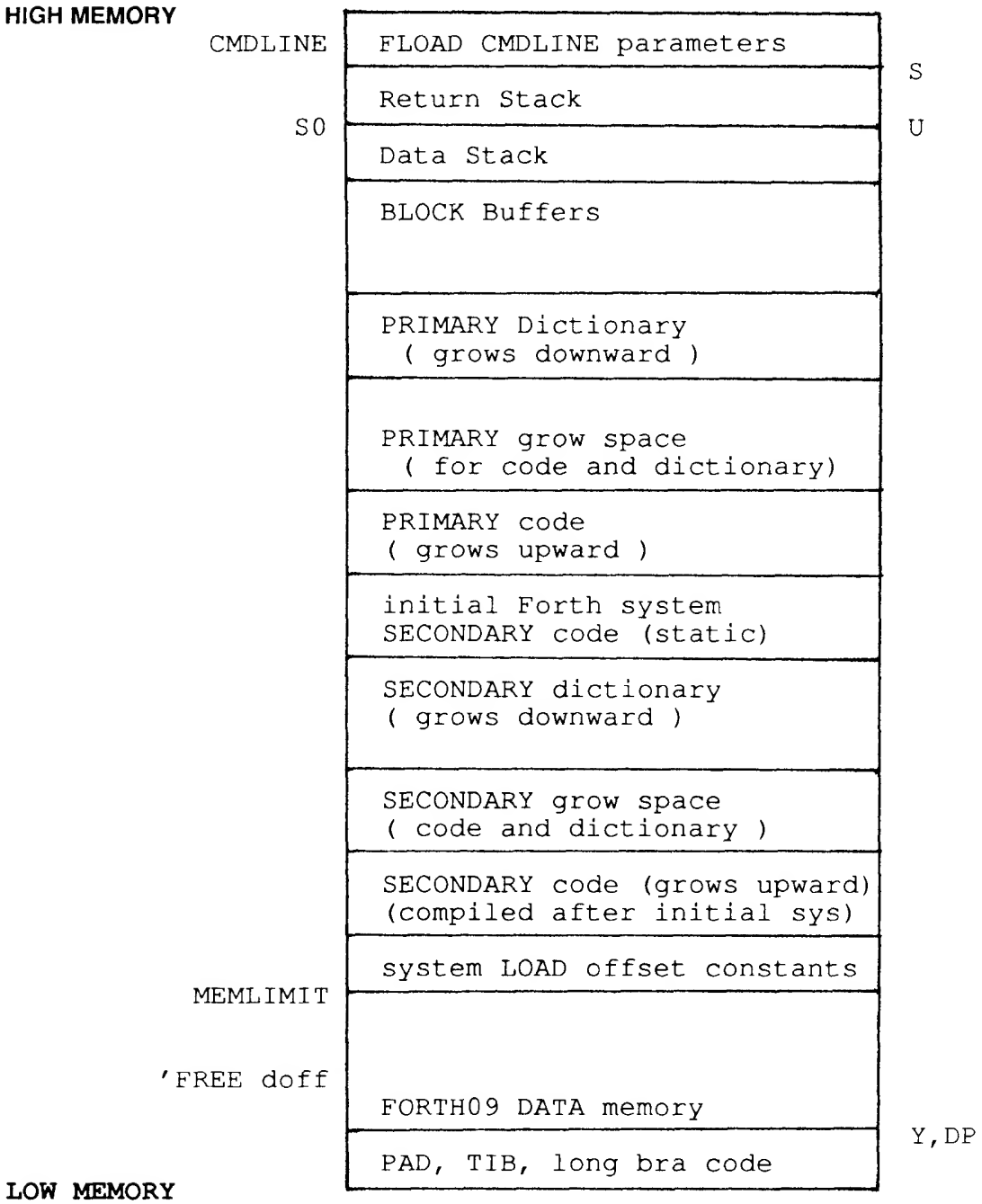
OPERATING SYSTEM INTERFACE

FORTH09 goes beyond the Forth-83 standard in providing an interface to the OS-9 operating system. Forth systems are often implemented without any operating system. In these cases Forth itself provides all control of the computer and peripherals, and mass storage consists entirely of the **BLOCK** storage without any file system. FORTH09 runs as a process under the OS-9 operating system, and **BLOCK** storage in FORTH09 is directed to OS-9 RBF files.

FORTH09 provides words implementing most of the OS-9 system calls. Any additional calls the user desires to use can be user implemented with little difficulty. A thorough study of the OS-9 technical manual should be undertaken to understand the power available through the system calls. A good place to start is with the I/O calls, for opening and creating "paths" and performing input and output.

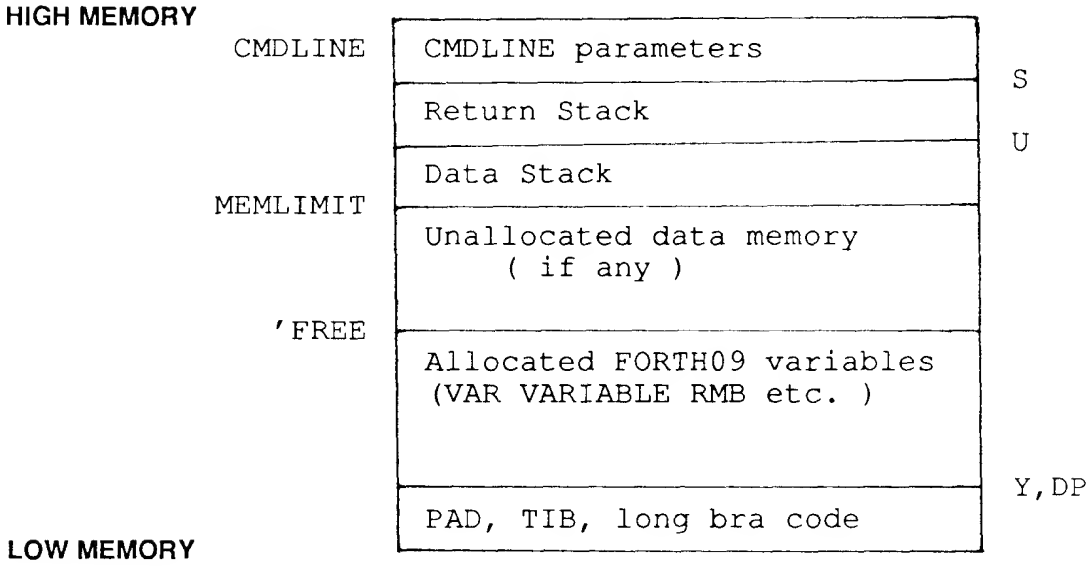
FORTH09 SYSTEM MEMORY MAP

(occupies FLOAD process data memory)



Application Runtime Data Memory Map

(of modules saved by SAVE)



DICTIONARY HEADER STRUCTURE

```
            csect
d.link      rmb 2       link address of next header in thread
d.hash      rmb 2       name hash code
d.prec      rmb 1       precedence and flag bits
d.complen rmb 1         code compilation length for inline
d.code      rmb 2       code offset from base address
d.chars     rmb 1       number of characters in name
d.name      rmb 1..n    name chars, n= lesser of d.chars or WIDTH
            endsect
```

The last name character stored has the high bit set. d.chars may be greater than the number of characters stored.

D.PREC DEFINITIONS

```
            csect
immed       equ $80     immediate word
componly    equ $01     word is compilation only word
secode      equ $02     word is SECONDARY word
secbase     equ $04     d.code offset is from secondary base area
inlineok    equ $08     may be compiled inline (code words only)
inline      equ $10     must be compiled inline ONLY
            endsect
```

Other bits are reserved for future optimization features.

Code Words and the number of bytes of code that will compile inline when **COMPLIMIT** is >= the listed number.

WORD	Bytes
!	6
#0	6
'FREE	6
*	35
+	6
+!	8
-	8
-ROT	9
0	4
1+	6
1-	7
2!	10
2*	4
2+	7
2-	7
2/	4
2@	8
2DROP	2
2DUP	6
2OVER	6
2ROT	29
2SWAP	16
<	17
><	6
?DUP	6
@	4
ABS	14
AND	8
BLANK	21
C!	6
C@	5
CMOVE	33
CMOVE>	37
COUNT	7
D!	10
D+	16
D2/	8
D@	8
DABS	24
DDROP	2
DDUP	6
DECIMAL	5
DNEGATE	20
DOVER	6
DROP	2
DROT	29
DSWAP	16
DUP	4
ERASE	24
EXECUTE	2
FALSE	4
FILL	17
NEGATE	10
NIP	4
NOT	4
OR	8
OVER	4
PICK	11
ROLL	22
ROT	9
S0	4
S>D	10
SHIFT	22
SP@	4
SWAP	6
SWAP!	4
TO	2
TRUE	5
TUCK	6
XOR	8

The following words are always compiled inline regardless of what value is in **COMPLIMIT**.

WORD	Bytes
>R	4
EXIT	1
I	4
J	4
R>	4
R@	4
RDROP	2

FORTH09 ERROR MESSAGES

All of the error messages listed cause the word **ABORT** to be executed unless otherwise noted. **ABORT** will clear the data and return stacks, and reset the input stream to the console, by setting the value of **BLK** to zero.

--0 MAYBE args There were no pairs of arguments supplied after **MAYBE**. This error is detected by **THENAGAIN**.

--bad instruction mode An invalid addressing mode for an ASSEMBLER word was specified.

--can't forget this deep An attempt was made to **FORGET** a word that was loaded with the original system be FLOAD. NOTE: This applies to systems save with **SAVESYS** also.

--can't SAVE secondary You tried to save a secondary word as an application module using **SAVE** or **XSAVE**.

--code imbalance **END-CODE** was executed without **CODE** or **;CODE** or else the stack has been altered since those words executed.

--code space overflow The word **ALLOT** was called with an argument that exceeded the remaining code space in the current code area (PRIMARY or SECONDARY).

--compilation only An attempt was made to execute a word in the interpret state that may only be used within a colon definition.

-- compiling secondary word to primary An attempt was made to compile a SECONDARY word into a PRIMARY word definition. This is not allowed because if the PRIMARY word is SAVEd as an application, the SECONDARY code will no longer be accessible to it.

--creating null word **CREATE** or **(CREATE)** have been executed with the input stream exhausted. If input is from the console, the new name must be on the same line as the word that makes the dictionary entry. E.g. **:** on a line by itself will cause this error.

--data memory overflow The word **ALLOT_D** was called with an argument that exceeded the remaining data memory. Caused by **VARIABLE** or **RMB** or another word that uses **ALLOT_D** .

--illegal addr for >RESOLVE or <RESOLVE The stack contents were probably altered in between control words using **>MARK <MARK >RESOLVE <RESOLVE** .

--incomplete.. forget me The word being defined was added to the dictionary, but the definition is incomplete and should not be executed. Use **FORGET** to remove the word from the dictionary. This may happen when defining a word using **CREATE** and **DOES>** and an error is encountered after the **DOES>** .

--invalid MAYBE structure The argument list for **MAYBE** was incorrect. (Should be pairs of arguments, a byte value literal followed by a word to be executed).

--memory full during compilation A colon definition was started with the current code memory area (PRIMARY or SECONDARY) nearly full. You may need to reload FORTH09 with FLOAD and reapportion memory.

--missing BEGIN **UNTIL** or **AGAIN** was encountered without **BEGIN** or else something has tampered with the stack.

--missing DO **LOOP** or **+LOOP** have been executed without executing **DO** , or else the stack contents have been altered since **DO** .

--missing GLOBAL The word **MODULE** was executed without **GLOBAL** or else something has tampered with the stack.

--missing IF **ELSE ENDIF** or **THEN** were executed without the **IF** statement, or else the stack contents have been altered since **IF** .

--missing LOCAL **GLOBAL** was encountered without **LOCAL** or else something has tampered with the stack.

--missing MAYBE **THENAGAIN** was encountered without **MAYBE** or else something has tampered with the stack.

--missing WHILE **REPEAT** was encountered without **WHILE** or else something has tampered with the stack.

--missing keyword? ; has been executed with an unexpected sys word on the stack, usually indicating improper usage of some IMMEDIATE word that manipulates the stack during compilation, or a missing control word such as **UNTIL ENDIF LOOP** etc.

--missing operand An ASSEMBLER word was missing an operand.

--not executing A word was used inside a colon definition that is allowed only in the execute state.

--range error An ASSEMBLER conditional required a branch out of range for the short relative branch.

--redefined THIS IS A WARNING ONLY that the last word added to the dictionary already existed in the search order.

-- stack overflow The stack has been over filled. (Items added to within 4 bytes of the stacks designated size).

-- stack underflow More items were removed from the stack than were there.

--unmatched conditional One of the ASSEMBLER control words required in pairs was missing. e.g. BEGIN without UNTIL.